



Linkers - Ian Taylor

Thu, 22 Aug 2024

Linkers - Ian Taylor

[Thu, 22 Aug 2024]

- [Linkers](#)

Linkers

- [Part 1 Introduction, personal history, first half of what's-a-linker](#)
- [Part 2 What's-a-linker: Dynamic linking, linker data types, linker operation](#)
- [Part 3 Address spaces, Object file formats](#)
- [Part 4 Shared Libraries](#)
- [Part 5 More Shared Libraries -- specifically, linker implementation; ELF Symbols](#)
- [Part 6 Relocations, Position Dependent Shared Libraries](#)
- [Part 7 Thread Local Storage \(TLS\) optimization](#)
- [Part 8 ELF Segments and Sections](#)
- [Part 9 Symbol Versions, Relaxation optimization](#)
- [Part 10 Parallel linking](#)
- [Part 11 Archive format](#)
- [Part 12 Symbol resolution](#)
- [Part 13 Symbol resolution from the user's point of view; Static Linking vs. Dynamic Linking](#)
- [Part 14 Link time optimization, aka Whole Program optimization; Initialization Code](#)

- [Part 15 COMDAT sections](#)
- [Part 16 C++ Template Instantiation, Exception Frames](#)
- [Part 17 Warning Symbols,](#)
- [Part 18 Incremental Linking](#)
- [Part 19 start and stop Symbols, Byte Swapping](#)
- [Part 20 Last post; Update on gold's status](#)

Airs – Ian Lance Taylor

Linkers part 1

I've been working on and off on a new linker. To my surprise, I've discovered in talking about this that some people, even some computer programmers, are unfamiliar with the details of the linking process. I've decided to write some notes about linkers, with the goal of producing an essay similar to my existing one about the GNU configure and build system.

As I only have the time to write one thing a day, I'm going to do this on my blog over time, and gather the final essay together later. I believe that I may be up to five readers, and I hope y'all will accept this digression into stuff that matters. I will return to random philosophizing and minding other people's business soon enough.

A Personal Introduction

Who am I to write about linkers?

I wrote my first linker back in 1988, for the AMOS operating system which ran on Alpha Micro systems. (If you don't understand the following description, don't worry; all will be explained below). I used a single global database to register all symbols. Object files were checked into the database after they

had been compiled. The link process mainly required identifying the object file holding the main function. Other objects files were pulled in by reference. I reverse engineered the object file format, which was undocumented but quite simple. The goal of all this was speed, and indeed this linker was much faster than the system one, mainly because of the speed of the database.

I wrote my second linker in 1993 and 1994. This linker was designed and prototyped by Steve Chamberlain while we both worked at Cygnus Support (later Cygnus Solutions, later part of Red Hat). This was a complete reimplement of the BFD based linker which Steve had written a couple of years before. The primary target was a.out and COFF. Again the goal was speed, especially compared to the original BFD based linker. On SunOS 4 this linker was almost as fast as running the cat program on the input .o files.

The linker I am now working, called gold, on will be my third. It is exclusively an ELF linker. Once again, the goal is speed, in this case being faster than my second linker. That linker has been significantly slowed down over the years by adding support for ELF and for shared libraries. This support was patched in rather than being designed in. Future plans for the new linker include support for incremental linking—which is another way of increasing speed.

There is an obvious pattern here: everybody wants linkers to be faster. This is because the job which a linker does is

uninteresting. The linker is a speed bump for a developer, a process which takes a relatively long time but adds no real value. So why do we have linkers at all? That brings us to our next topic.

A Technical Introduction

What does a linker do?

It's simple: a linker converts object files into executables and shared libraries. Let's look at what that means. For cases where a linker is used, the software development process consists of writing program code in some language: e.g., C or C++ or Fortran (but typically not Java, as Java normally works differently, using a loader rather than a linker). A compiler translates this program code, which is human readable text, into another form of human readable text known as assembly code. Assembly code is a readable form of the machine language which the computer can execute directly. An assembler is used to turn this assembly code into an *object file*. For completeness, I'll note that some compilers include an assembler internally, and produce an object file directly. Either way, this is where things get interesting.

In the old days, when dinosaurs roamed the data centers, many programs were complete in themselves. In those days there was generally no compiler—people wrote directly in assembly code—and the assembler actually generated an *executable file* which the machine could execute directly. As languages like Fortran

and Cobol started to appear, people began to think in terms of libraries of subroutines, which meant that there had to be some way to run the assembler at two different times, and combine the output into a single executable file. This required the assembler to generate a different type of output, which became known as an object file (I have no idea where this name came from). And a new program was required to combine different object files together into a single executable. This new program became known as the linker (the source of this name should be obvious).

Linkers still do the same job today. In the decades that followed, one new feature has been added: shared libraries.

More tomorrow.

Posted August 22, 2007 in [Programming](#)
by Ian Lance Taylor

Tags:

Comments

23 responses to “Linkers part 1”



[rmathew](#)

[August 23, 2007](#)

I am looking forward to the rest of this series. I hope you will also touch upon sorting out template instantiations, doing link-time optimisations, etc. that put additional burdens on a linker making it slower, though also more useful.

PS: I'm glad that you have started blogging regularly. I like it that you seem to have thought through most of the issues that you blog about, even if at times I don't find myself agreeing with your conclusions.



[ncm](#)

[August 23, 2007](#)

I too am looking forward to the rest of the series. It seems to me that it's getting hard to make something recognizable, any more, as a classical linker, now that code generation and optimization are essential parts of the job.



[movement](#)

[August 23, 2007](#)

Linkers and related topics such as runtime loaders and shared libraries are a mystery at some level to most programmers, I find. It initially took

me a lot of time at staring at assembly output to work out link-time relocations actually worked: both the Solaris linker and the GNU one are pretty impenetrable if you're just browsing. It'll be very interesting to hear some details from you on these topics.

Some related reading:

Sun's Linkers and Libraries Guide

<http://blogs.sun.com/rie/>

<http://blogs.sun.com/msw/>

<http://blogs.sun.com/ali/>

John Levine's old book Linkers and Loaders too. I didn't get much out of this book; unfortunately it was pretty outdated, and not very clearly put together. I suspect the exercises would prove interesting to do though.



[Ian Lance Taylor](#)

[August 23, 2007](#)

Thanks for the notes.

I tend to view template instantiation and link-time optimizations as separate from the linker proper. In implementations I know about, they are done before invoking the linker itself, or they are done via plugins which the linker invokes. That is, under the hood, there is still a classical linker.

But since there is interest, perhaps I will move on to those topics after covering the linker proper.



[Ivan](#)

[August 23, 2007](#)

Great Idea,

I look forward to reading these entries.

Thanks,

Ivan Novick

<http://www.0x4849.net>

[christian schorn » Blog Archive » links for 2007-08-30](#)

[August 30, 2007](#)

[...] [Airs – Ian Lance Taylor » Linkers part 1 \(tags: programming basics\) \[...\]](#)

[Mark J. Wielaard » Ian Lance Taylor's Linker Notes](#)

[August 31, 2007](#)

[...] [Linkers part 1 – A Personal Introduction and A Technical Introduction. \[...\]](#)



[jrlevine](#)

[September 13, 2007](#)

Nice series. Believe it or not, relocating loaders predate assemblers, with the first one in the late 1940s, and linking loaders aren't much later. This technology goes way back.

Also, I was kind of surprised at the comment that my books was outdated. One of the reasons I wrote it was that linker technology changes so slowly. There hasn't been an interesting new idea since incremental linkers about 20 years ago, knowledge of linkers has been mostly programmer folklore, so I figured I'd write it down so it'd be at last available somewhere. The descriptions of ELF, ECOFF, and the way they support dynamic linking are as far as I know still current, nothing's changed since I wrote the book in 2000.



[Ian Lance Taylor](#)

[September 13, 2007](#)

Thanks for the note. There is a lot I don't know about the history.

I didn't make the comment about your book myself; it's certainly the best description of linkers I know of. Still, unless I misremember, there are some recent important ideas which

aren't covered, such as ELF symbol versions, ELF (and Mach-O) symbol visibility, interposition with LD_PRELOAD and the like, TLS details. I don't actually have a copy to hand, so I hope I am not misrepresenting it. These are not major ideas like incremental linking, but they are things which the relatively few people who work with linkers need to understand.



avjo

[November 4, 2007](#)

Hi Ian,

This series is so educating and interesting ! Thank you for that !

Just wondering here... will your new linker be GPLv2 or GPLv3 ?

~avjo



[Ian Lance Taylor](#)

[November 5, 2007](#)

The goal is for the new linker to be part of the GNU binutils, which means that it will be GPLv3.

(I'll add that I think that in practice there is very little difference between GPLv2 and GPLv3.)

[E. Huntley – Programming & Development Blog » Blog Archive » Back to the “Basics”](#)
[February 23, 2009](#)

[...] The entries start here. And continue through his entry archives to mid September 2007. I highly recommend giving at least the first few entries a quick read-through if you are like me, and want a better understanding of the development tools we use every day. [...]

[zur::Linux » Gold Linker](#)
[September 1, 2011](#)

[...] If you want to know more about linkers and Gold in particular Ian Lance Taylor has a twenty-part series about linker internals on his blog. [...]

[Yearzero.flaminghorns.com – September is Linker Month!!!](#)
[August 27, 2013](#)

[...] There are around 20 odd articles to be read. The link series for the article is as follows: <http://www.airs.com/blog/archives/38> – <http://www.airs.com/blog/archives/57> Tagged and categorized as: General | No comments [...]

[Relocatable objects – BYTEC/16](#)
[December 16, 2013](#)

[...] code generation, and that's exactly what symbol tables and relocation tables are used for. I used this material by Ian Lance Taylor to understand the basics of [...]

[What is PLT/GOT? | CL-UAT](#)
[December 23, 2014](#)

[...] Also, Ian Lance Taylor, the author of GOLD has put up an article series on his blog which is totally worth reading (twenty parts!): entry point here "Linkers part 1". [...]

[A journey into Radare 2 – Part 2: Exploitation – Megabeets](#)
[September 3, 2017](#)

[...] the function address from the GOT. To read more about the linking process, I highly recommend this series of articles about linkers by Ian Lance [...]

[????????? ???? ???? , ?? : ??? ????? ????? | Linux /etc/](#)
[April 5, 2018](#)

[...] – process init functions flow
<http://www.airs.com/blog/archives/38> – linkers (20 parts article!!!) – nice to have [...]

[Ce este gold linker?](#)
[April 27, 2021](#)

[...] diferitelor linkeri GNU, care explic? multe dintre deciziile de proiectare care au dus la gold. El scrie [...]

[What is the gold linker? – PhotoLens](#)

[November 5, 2021](#)

[...] of the various GNU linkers, which explains many of the design decisions leading up to gold. He writes [...]

[x86 – ¿Qué es PLT / GOT? – CodeBug](#)

[December 30, 2021](#)

[...] una serie de artículos en su blog que vale la pena leer (¡veinte partes!): Punto de entrada aquí "Linkers part 1" [...]

[Replacing ld with gold – any experience? – PhotoLens](#)

[February 21, 2022](#)

[...] promises to be much faster than ld, so it may help speeding up test cycles for large C++ applications, but [...]

[Exploring GNU Gold Linker | Baeldung on Linux](#)

[November 17, 2023](#)

[...] gold is an ELF linker developed from scratch in C++ by Ian Lance Taylor. It was released in March 2008 and included in GNU [...]

Leave a Reply

You must be [logged in](#) to post a comment.

Airs – Ian Lance Taylor

Proudly powered by [WordPress](#)

This article was downloaded by **calibre** from <http://www.airs.com/blog/archives/38>

| [Section menu](#) | [Main menu](#) |

Airs – Ian Lance Taylor

Linkers part 2

I'm back, and I'm still doing the linker technical introduction.

Shared libraries were invented as an optimization for virtual memory systems running many processes simultaneously. People noticed that there is a set of basic functions which appear in almost every program. Before shared libraries, in a system which runs multiple processes simultaneously, that meant that almost every process had a copy of exactly the same code. This suggested that on a virtual memory system it would be possible to arrange that code so that a single copy could be shared by every process using it. The virtual memory system would be used to map the single copy into the address space of each process which needed it. This would require less physical memory to run multiple programs, and thus yield better performance.

I believe the first implementation of shared libraries was on SVR3, based on COFF. This implementation was simple, and basically assigned each shared library a fixed portion of the virtual address space. This did not require any significant changes to the linker. However, requiring each shared library to

reserve an appropriate portion of the virtual address space was inconvenient.

SunOS4 introduced a more flexible version of shared libraries, which was later picked up by SVR4. This implementation postponed some of the operation of the linker to runtime. When the program started, it would automatically run a limited version of the linker which would link the program proper with the shared libraries. The version of the linker which runs when the program starts is known as the *dynamic linker*. When it is necessary to distinguish them, I will refer to the version of the linker which creates the program as the *program linker*. This type of shared libraries was a significant change to the traditional program linker: it now had to build linking information which could be used efficiently at runtime by the dynamic linker.

That is the end of the introduction. You should now understand the basics of what a linker does. I will now turn to how it does it.

Basic Linker Data Types

The linker operates on a small number of basic data types: *symbols*, *relocations*, and *contents*. These are defined in the input object files. Here is an overview of each of these.

A symbol is basically a name and a value. Many symbols represent static objects in the original source code—that is, objects which exist in a single place for the duration of the

program. For example, in an object file generated from C code, there will be a symbol for each function and for each global and static variable. The value of such a symbol is simply an offset into the contents. This type of symbol is known as a *defined* symbol. It's important not to confuse the value of the symbol representing the variable `my_global_var` with the value of `my_global_var` itself. The value of the symbol is roughly the address of the variable: the value you would get from the expression `&my_global_var` in C.

Symbols are also used to indicate a reference to a name defined in a different object file. Such a reference is known as an *undefined* symbol. There are other less commonly used types of symbols which I will describe later.

During the linking process, the linker will assign an address to each defined symbol, and will *resolve* each undefined symbol by finding a defined symbol with the same name.

A relocation is a computation to perform on the contents. Most relocations refer to a symbol and to an offset within the contents. Many relocations will also provide an additional operand, known as the *addend*. A simple, and commonly used, relocation is "set this location in the contents to the value of this symbol plus this addend." The types of computations that relocations do are inherently dependent on the architecture of the processor for which the linker is generating code. For example, RISC processors which require two or more

instructions to form a memory address will have separate relocations to be used with each of those instructions; for example, “set this location in the contents to the lower 16 bits of the value of this symbol.”

During the linking process, the linker will perform all of the relocation computations as directed. A relocation in an object file may refer to an undefined symbol. If the linker is unable to resolve that symbol, it will normally issue an error (but not always: for some symbol types or some relocation types an error may not be appropriate).

The contents are what memory should look like during the execution of the program. Contents have a size, an array of bytes, and a type. They contain the machine code generated by the compiler and assembler (known as *text*). They contain the values of initialized variables (*data*). They contain static unnamed data like string constants and switch tables (read-only data or *rdata*). They contain uninitialized variables, in which case the array of bytes is generally omitted and assumed to contain only zeroes (*bss*). The compiler and the assembler work hard to generate exactly the right contents, but the linker really doesn't care about them except as raw data. The linker reads the contents from each file, concatenates them all together sorted by type, applies the relocations, and writes the result into the executable file.

Basic Linker Operation

At this point we already know enough to understand the basic steps used by every linker.

- Read the input object files. Determine the length and type of the contents. Read the symbols.
- Build a symbol table containing all the symbols, linking undefined symbols to their definitions.
- Decide where all the contents should go in the output executable file, which means deciding where they should go in memory when the program runs.
- Read the contents data and the relocations. Apply the relocations to the contents. Write the result to the output file.
- Optionally write out the complete symbol table with the final values of the symbols.

More tomorrow.

Posted August 23, 2007 in [Programming](#)
by Ian Lance Taylor

Tags:

Comments

14 responses to “Linkers part 2”



[ncm](#)

[August 27, 2007](#)

Apollo Aegis had shared libraries right from the beginning, in 1981. (Aegis heritage was usually traced directly to Multics, bypassing Unix.) Each shared library was assigned a fixed position in the address space, and each symbol given a fixed global 16-bit ID and entry in a table. Big programs and libraries tended to crowd that table, so Mentor Graphics had to poke out such unused C library functions as qsort.



[Ian Lance Taylor](#)

[August 27, 2007](#)

Thanks for the info. I used Apollo Aegis systems around 1985, but I only programmed them in T, never in any compiled language. From the description, it seems pretty similar to SVR3 shared libraries.



[ncm](#)

[August 28, 2007](#)

Apollo Aegis was interesting in a number of other ways. All code, shared library or main program, were directly mapped (a la mmap), so the code was all concentrated at the beginning of the file, with annotations after. Page faults for code loaded via the net were satisfied via the net. The file system expanded environment variables in symbolic link text, a feature since

implemented (to my knowledge, only) in DGUX and Dragonfly BSD. Apollo's DSEE version control (and build) system became the basis for ClearCase. Apollo's ACL permissions system and its remote procedure call apparatus was the basis for much of DCE and, thence, Microsoft CIFS.

[Mark J. Wielaard » Ian Lance Taylor's Linker Notes](#)
[August 31, 2007](#)

[...] Linkers part 2 – Linker Technical Introduction, Basic Linker Data Types, Basic Linker Operations. [...]



Joe Buck

[September 12, 2007](#)

Dec's VMS had shared libraries from the beginning, in 1980. I was a VMS user before I was a Unix user (and before that used DEC's RSX-11 and RT-11), and was regularly stunned when I went to grad school at UC Berkeley at all of the grad students who thought that any given computer science concept was invented when it first appeared in some flavor of Unix.



Joe Buck

[September 12, 2007](#)

Also, I recall that the VMS shared library implementation was quite modern, in the sense that it used position-independent code and mapped shared libraries into address spaces at any point. The page size was tiny, however, only 512 bytes. But then, a whole department would share one massive Vax 11/780 with a whopping 2 Mb of memory back when I started.



[Ian Lance Taylor](#)

[September 13, 2007](#)

Thanks for the note. I used TOPS-20 some in high school, but I never used VMS very much.



[jrlevine](#)

[September 13, 2007](#)

I think you'll find that shared libraries go back to Multics in about 1969. The Multics machines had hardware segments (sort of like the 286's but bigger) with each segment dynamically linking to others. Before that, executables with shared code were common, so that if three people were using the text editor they'd all share the same copy of the code, but each program had all of the libraries it needed linked into it. The PDP-6 timesharing system had that probably by 1965, early PDP-11 Unix by 1972 or 3. Again, these ideas go way, way back.



rskrishnan

[October 8, 2007](#)

I think shared libraries have been around for a while. I have used VMS systems with some very sophisticated linkers and loaders. I've also "seen" as/400 systems (but their terminology is totally gibberish to me).

I really liked assemblers and linkers as discussed in "Systems Programming" by John J. Donovan. Very exhaustive discussion on what happens at each stage of the compile/link/load/execute process. Very informative and assumes very little knowledge going in.



[Ian Lance Taylor](#)

[October 8, 2007](#)

Thanks for the note. Maybe some day I'll research these earlier systems.

Thanks for the pointer to Donovan's book; I'm not familiar with it.



[Jessica Hamilton](#)

[October 12, 2008](#)

Hi Ian,

Your series on linkers has been interesting and informative. Especially since I am attempting (seemingly in vain) to write a very basic linker for ELF. No shared libraries or other strange stuff. Just combine objects and static libraries into a basic executable.

I understand the basics of how it works, but haven't really been able to figure out how to piece it together. I can parse objects and libraries, parse symbol tables, and print out most of this sort of information.

I am hoping you could help me with a bit more of a breakdown of how to put this altogether. Something kind of like pseudo-code.

Many Thanks,

Jessica



[Ian Lance Taylor](#)

[October 13, 2008](#)

Thanks for the note. I recommend that you look at the source code for the linker I wrote, gold. It's even better than pseudo-code: it's real code.

To get a working linker concentrate on building the symbol table, placing the input sections in the output file, and applying relocations.



[Jessica Hamilton](#)

[October 13, 2008](#)

Thanks Ian. Hopefully I can scrape the mechanics of linking out of all the multi-threading and syntax in general! My eyes tend to glaze over with C/C++, but maybe I'll get there.

Pseudo-code does have the advantage of simplicity over real code, btw 😊



berkus

[January 2, 2009](#)

Jessica, I'm probably too late for the party, but take a look at this simple ELF linker:

<http://bzt.madfire.net/odin/files/head%3A/tools/sjofn/>

Leave a Reply

You must be [logged in](#) to post a comment.

[Airs – Ian Lance Taylor](#)

Proudly powered by [WordPress](#)

This article was downloaded by **calibre** from <http://www.airs.com/blog/archives/39>

| [Section menu](#) | [Main menu](#) |

Airs – Ian Lance Taylor

Linkers part 3

Continuing notes on linkers.

Address Spaces

An address space is simply a view of memory, in which each byte has an address. The linker deals with three distinct types of address space.

Every input object file is a small address space: the contents have addresses, and the symbols and relocations refer to the contents by addresses.

The output program will be placed at some location in memory when it runs. This is the output address space, which I generally refer to as using *virtual memory addresses*.

The output program will be loaded at some location in memory. This is the *load memory address*. On typical Unix systems virtual memory addresses and load memory addresses are the same. On embedded systems they are often different; for example, the initialized data (the initial contents of global or static variables) may be loaded into ROM at the load memory address, and then copied into RAM at the virtual memory address.

Shared libraries can normally be run at different virtual memory address in different processes. A shared library has a *base address* when it is created; this is often simply zero. When the dynamic linker copies the shared library into the virtual memory space of a process, it must apply relocations to adjust the shared library to run at its virtual memory address. Shared library systems minimize the number of relocations which must be applied, since they take time when starting the program.

Object File Formats

As I said above, an assembler turns human readable assembly language into an object file. An object file is a binary data file written in a format designed as input to the linker. The linker generates an executable file. This executable file is a binary data file written in a format designed as input for the operating system or the loader (this is true even when linking dynamically, as normally the operating system loads the executable before invoking the dynamic linker to begin running the program). There is no logical requirement that the object file format resemble the executable file format. However, in practice they are normally very similar.

Most object file formats define *sections*. A section typically holds memory contents, or it may be used to hold other types of data. Sections generally have a name, a type, a size, an address, and an associated array of data.

Object file formats may be classed in two general types: record oriented and section oriented.

A record oriented object file format defines a series of records of varying size. Each record starts with some special code, and may be followed by data. Reading the object file requires reading it from the beginning and processing each record. Records are used to describe symbols and sections. Relocations may be associated with sections or may be specified by other records. IEEE-695 and Mach-O are record oriented object file formats used today.

In a section oriented object file format the file header describes a section table with a specified number of sections. Symbols may appear in a separate part of the object file described by the file header, or they may appear in a special section. Relocations may be attached to sections, or they may appear in separate sections. The object file may be read by reading the section table, and then reading specific sections directly. ELF, COFF, PE, and a.out are section oriented object file formats.

Every object file format needs to be able to represent debugging information. Debugging information is generated by the compiler and read by the debugger. In general the linker can just treat it like any other type of data. However, in practice the debugging information for a program can be larger than the actual program itself. The linker can use various techniques to reduce the amount of debugging information, thus reducing the

size of the executable. This can speed up the link, but requires the linker to understand the debugging information.

The a.out object file format stores debugging information using special strings in the symbol table, known as stabs. These special strings are simply the names of symbols with a special type. This technique is also used by some variants of ECOFF, and by older versions of Mach-O.

The COFF object file format stores debugging information using special fields in the symbol table. This type information is limited, and is completely inadequate for C++. A common technique to work around these limitations is to embed stabs strings in a COFF section.

The ELF object file format stores debugging information in sections with special names. The debugging information can be stabs strings or the DWARF debugging format.

More next week.

Posted August 24, 2007 in [Programming](#)
by Ian Lance Taylor

Tags:

Comments

3 responses to “Linkers part 3”

[Mark J. Wielaard » Ian Lance Taylor's Linker Notes](#)

[August 31, 2007](#)

[...] Linkers part 3 – Address Spaces, Object File Formats. [...]



abhi

[March 3, 2014](#)

Hi Ian,

I have a quick query. So once the dynamic linker has loaded program executable and all the shared libraries required by it. Does Linker keeps track of different Section Header for the elf executable and the shared libraries loaded.

Actually I am modifying the libc6 dynamic linker and I need the start and end/length for each of the plt sections.

The link map maintained by the linker only has information about the program headers. (In this case plt and some other sections are clubbed together into a single load section).

Does linker maintains some data structure or pointer through which I can get the address range for each of the respective plt section in memory?

Thanks



[Ian Lance Taylor](#)

[June 6, 2014](#)

Given the program headers, you can find everything else. The PT_DYNAMIC tag points you to the dynamic section. In there the DT_JMPREL tag should get you to the PLT. The details are target dependent.

Leave a Reply

You must be [logged in](#) to post a comment.

Airs – Ian Lance Taylor

Proudly powered by [WordPress](#)

This article was downloaded by **calibre** from <http://www.airs.com/blog/archives/40>

| [Section menu](#) | [Main menu](#) |

Airs – Ian Lance Taylor

Linkers part 4

Shared Libraries

We've talked a bit about what object files and executables look like, so what do shared libraries look like? I'm going to focus on ELF shared libraries as used in SVR4 (and GNU/Linux, etc.), as they are the most flexible shared library implementation and the one I know best.

Windows shared libraries, known as DLLs, are less flexible in that you have to compile code differently depending on whether it will go into a shared library or not. You also have to express symbol visibility in the source code. This is not inherently bad, and indeed ELF has picked up some of these ideas over time, but the ELF format makes more decisions at link time and is thus more powerful.

When the program linker creates a shared library, it does not yet know which virtual address that shared library will run at. In fact, in different processes, the same shared library will run at different address, depending on the decisions made by the dynamic linker. This means that shared library code must be *position independent*. More precisely, it must be position independent after the dynamic linker has finished loading it. It is

always possible for the dynamic linker to convert any piece of code to run at any virtual address, given sufficient relocation information. However, performing the reloc computations must be done every time the program starts, implying that it will start more slowly. Therefore, any shared library system seeks to generate position independent code which requires a minimal number of relocations to be applied at runtime, while still running at close to the runtime efficiency of position dependent code.

An additional complexity is that ELF shared libraries were designed to be roughly equivalent to ordinary archives. This means that by default the main executable may override symbols in the shared library, such that references in the shared library will call the definition in the executable, even if the shared library also defines that same symbol. For example, an executable may define its own version of `malloc`. The C library also defines `malloc`, and the C library contains code which calls `malloc`. If the executable defines `malloc` itself, it will override the function in the C library. When some other function in the C library calls `malloc`, it will call the definition in the executable, not the definition in the C library.

There are thus different requirements pulling in different directions for any specific ELF implementation. The right implementation choices will depend on the characteristics of the processor. That said, most, but not all, processors make fairly

similar decisions. I will describe the common case here. An example of a processor which uses the common case is the i386; an example of a processor which make some different decisions is the PowerPC.

In the common case, code may be compiled in two different modes. By default, code is position dependent. Putting position dependent code into a shared library will cause the program linker to generate a lot of relocation information, and cause the dynamic linker to do a lot of processing at runtime. Code may also be compiled in position independent mode, typically with the `-fpic` option. Position independent code is slightly slower when it calls a non-static function or refers to a global or static variable. However, it requires much less relocation information, and thus the dynamic linker will start the program faster.

Position independent code will call non-static functions via the *Procedure Linkage Table* or *PLT*. This PLT does not exist in `.o` files. In a `.o` file, use of the PLT is indicated by a special relocation. When the program linker processes such a relocation, it will create an entry in the PLT. It will adjust the instruction such that it becomes a PC-relative call to the PLT entry. PC-relative calls are inherently position independent and thus do not require a relocation entry themselves. The program linker will create a relocation for the PLT entry which tells the dynamic linker which symbol is associated with that entry. This process reduces the

number of dynamic relocations in the shared library from one per function call to one per function called.

Further, PLT entries are normally relocated lazily by the dynamic linker. On most ELF systems this laziness may be overridden by setting the `LD_BIND_NOW` environment variable when running the program. However, by default, the dynamic linker will not actually apply a relocation to the PLT until some code actually calls the function in question. This also speeds up startup time, in that many invocations of a program will not call every possible function. This is particularly true when considering the shared C library, which has many more function calls than any typical program will execute.

In order to make this work, the program linker initializes the PLT entries to load an index into some register or push it on the stack, and then to branch to common code. The common code calls back into the dynamic linker, which uses the index to find the appropriate PLT relocation, and uses that to find the function being called. The dynamic linker then initializes the PLT entry with the address of the function, and then jumps to the code of the function. The next time the function is called, the PLT entry will branch directly to the function.

Before giving an example, I will talk about the other major data structure in position independent code, the *Global Offset Table* or *GOT*. This is used for global and static variables. For every reference to a global variable from position independent code,

the compiler will generate a load from the GOT to get the address of the variable, followed by a second load to get the actual value of the variable. The address of the GOT will normally be held in a register, permitting efficient access. Like the PLT, the GOT does not exist in a .o file, but is created by the program linker. The program linker will create the dynamic relocations which the dynamic linker will use to initialize the GOT at runtime. Unlike the PLT, the dynamic linker always fully initializes the GOT when the program starts.

For example, on the i386, the address of the GOT is held in the register %ebx. This register is initialized at the entry to each function in position independent code. The initialization sequence varies from one compiler to another, but typically looks something like this:

```
call __i686.get_pc_thunk.bx
add $offset,%ebx
```

The function __i686.get_pc_thunk.bx simply looks like this:

```
mov (%esp),%ebx
ret
```

This sequence of instructions uses a position independent sequence to get the address at which it is running. Then it uses

an offset to get the address of the GOT. Note that this requires that the GOT always be a fixed offset from the code, regardless of where the shared library is loaded. That is, the dynamic linker must load the shared library as a fixed unit; it may not load different parts at varying addresses.

Global and static variables are now read or written by first loading the address via a fixed offset from `%ebx`. The program linker will create dynamic relocations for each entry in the GOT, telling the dynamic linker how to initialize the entry. These relocations are of type `GLOB_DAT`.

For function calls, the program linker will set up a PLT entry to look like this:

```
jmp *offset(%ebx)
pushl #index
jmp first_plt_entry
```

The program linker will allocate an entry in the GOT for each entry in the PLT. It will create a dynamic relocation for the GOT entry of type `JMP_SLOT`. It will initialize the GOT entry to the base address of the shared library plus the address of the second instruction in the code sequence above. When the dynamic linker does the initial lazy binding on a `JMP_SLOT` reloc, it will simply add the difference between the shared library load address and the shared library base address to the GOT entry.

The effect is that the first `jmp` instruction will jump to the second instruction, which will push the index entry and branch to the first PLT entry. The first PLT entry is special, and looks like this:

```
pushl 4(%ebx)
jmp *8(%ebx)
```

This references the second and third entries in the GOT. The dynamic linker will initialize them to have appropriate values for a callback into the dynamic linker itself. The dynamic linker will use the index pushed by the first code sequence to find the `JMP_SLOT` relocation. When the dynamic linker determines the function to be called, it will store the address of the function into the GOT entry references by the first code sequence. Thus, the next time the function is called, the `jmp` instruction will branch directly to the right code.

That was a fast pass over a lot of details, but I hope that it conveys the main idea. It means that for position independent code on the i386, every call to a global function requires one extra instruction after the first time it is called. Every reference to a global or static variable requires one extra instruction. Almost every function uses four extra instructions when it starts to initialize `%ebx` (leaf functions which do not refer to any global variables do not need to initialize `%ebx`). This all has some negative impact on the program cache. This is the runtime

performance penalty paid to let the dynamic linker start the program quickly.

On other processors, the details are naturally different. However, the general flavour is similar: position independent code in a shared library starts faster and runs slightly slower.

More tomorrow.

Posted August 27, 2007 in [Programming](#)
by Ian Lance Taylor

Tags:

Comments

12 responses to “Linkers part 4”



wh5a

[August 29, 2007](#)

Thanks for your great article. I've got one question:

It seems if a pic function only access global variables but does not call global functions, it will call `__i686.get_pc_thunk.cx` to compute the GOT address, and its value will be cached in `%ecx`, instead of `%ebx`. Why is that?

I'm running Linux. Thanks.



[Ian Lance Taylor](#)

[August 29, 2007](#)

%ebx is a callee saved register for the i386, which means that if a function changes %ebx, it must save it at the start of a function and restore it at the end. This is normally the best choice for the GOT register, since it means that the value does not have to be recomputed or restored after a function call.

However, if a function does not call any other functions (i.e., it is a leaf function), then it is not important to keep the address of the GOT in a callee saved register. In fact, in that case, it is better to keep it in a caller saved register—that is, a register which a function is permitted to change without needing to save and restore it. So gcc optimizes by putting the GOT table in a caller saved register in a leaf function.

gcc does not always use %ecx, incidentally, though that is a common choice. Depending on the function, it may choose any available caller saved register.

[Mark J. Wielaard » Ian Lance Taylor's Linker Notes](#)

[August 31, 2007](#)

[...] Linkers part 4 – Shared Libraries (Procedure Linkage Table – PLT and Global Offset Table – GOT). [...]



[jrlevine](#)

[September 13, 2007](#)

The other advantage of PIC is better code sharing. If there's no relocation fixups in a page, all processes can share the same physical copy of the page. As soon as there's load time fixup, you need a separate copy of the page per process. Making and maintaining the copy is way more work than the fixups themselves, since it requires a trap to the system and a copy of the whole page.



[Ian Lance Taylor](#)

[September 13, 2007](#)

Thanks—I remembered to put that bit into part 6.



[jlh](#)

[October 28, 2008](#)

For the uneducated reader, it may worth saying explicitly that the offset added to the ebx register is the difference between the start of the GOT and the actual location in the code. Otherwise, the following sentence may not be as clear as you might think: “this requires that the GOT always be a fixed offset from the code, regardless of where the shared library is loaded”.

An interesting note is that these offsets are all fixed in the code at link time by the linker program.



[Ian Lance Taylor](#)

[October 28, 2008](#)

jlh: yes; thanks for the note.



berkus

[January 2, 2009](#)

Thanks, all this GOT/PLT thing became a bit more clear now. I was seeing the general picture before, but these little details is what was buzzing in my head all the time.



telenn

[June 14, 2013](#)

“For every reference to a global variable from position independent code, the compiler will generate a load from the GOT to get the address of the variable, followed by a second load to get the actual value of the variable.” ... “Every reference to a global or static variable requires one extra instruction”.

Well, I thought there was a difference between global and static variables, as explained by U.Drepper in his document “How to write shared libraries” :

For a non-static global variable (globvar) :

```
movl globalvar@GOT(%ebx), %edx
```

```
movl (%edx), %eax
```

For a static global variable :

```
movl staticvar@GOTOFF(%ebx), %eax
```

So it looks there's one instruction less for accessing a static global variable. It's as if the GOT entry for staticvar were a place for the variable itself, rather than a place for the absolute address of staticvar.

What do you think ?



telenn

[June 14, 2013](#)

“For every reference to a global variable from position independent code, the compiler will generate a load from the GOT to get the address of the variable, followed by a second load to get the actual value of the variable.” ... “Every reference to a global or static variable requires one extra instruction”.

Well, I thought there was a difference between global and static variables, as explained by U.Drepper in his document “How to write shared libraries” :

For a non-static global variable (globvar) :

```
movl globalvar@GOT(%ebx), %edx
```

```
movl (%edx), %eax
```

For a static global variable :

```
movl staticvar@GOTOFF(%ebx), %eax
```

So it looks there's one instruction less for accessing a static variable. It's as if the GOT entry for staticvar were a place for the variable itself, rather than a place for the absolute address of staticvar.

What do you think ?



[Ian Lance Taylor](#)

[July 12, 2013](#)

You're right, on some platforms the compiler can treat a static variable (or a variable with hidden visibility) differently and more efficiently. When this is done, a static variable does not require a GOT entry. The GOTOFF relocation computes the offset from the start of the GOT to the variable itself. This can work because there is no possibility that the variable is overridden by some other shared library, so the offset from the GOT to the variable is fixed.

[ltrace for RHEL 6 and 7 | Red Hat Developer Blog](#)

[July 11, 2014](#)

[...] to implement calls to shared libraries—procedure linkage tables, or PLT's. Ian Lance Taylor published a good treatment of the way dynamic linking works, for us the necessary thing is that inter-library [...]

Leave a Reply

You must be [logged in](#) to post a comment.

Airs – Ian Lance Taylor

Proudly powered by [WordPress](#)

This article was downloaded by **calibre** from <http://www.airs.com/blog/archives/41>

| [Section menu](#) | [Main menu](#) |

Airs – Ian Lance Taylor

Linkers part 5

Shared Libraries Redux

Yesterday I talked about how shared libraries work. I realized that I should say something about how linkers implement shared libraries. This discussion will again be ELF specific.

When the program linker puts position dependent code into a shared library, it has to copy more of the relocations from the object file into the shared library. They will become dynamic relocations computed by the dynamic linker at runtime. Some relocations do not have to be copied; for example, a PC relative relocation to a symbol which is local to shared library can be fully resolved by the program linker, and does not require a dynamic reloc. However, note that a PC relative relocation to a global symbol does require a dynamic relocation; otherwise, the main executable would not be able to override the symbol. Some relocations have to exist in the shared library, but do not need to be actual copies of the relocations in the object file; for example, a relocation which computes the absolute address of symbol which is local to the shared library can often be replaced with a `RELATIVE` reloc, which simply directs the dynamic linker to add the difference between the shared library's load address and its base

address. The advantage of using a `RELATIVE` reloc is that the dynamic linker can compute it quickly at runtime, because it does not require determining the value of a symbol.

For position independent code, the program linker has a harder job. The compiler and assembler will cooperate to generate special relocs for position independent code. Although details differ among processors, there will typically be a `PLT` reloc and a `GOT` reloc. These relocs will direct the program linker to add an entry to the `PLT` or the `GOT`, as well as performing some computation. For example, on the `i386` a function call in position independent code will generate a `R_386_PLT32` reloc. This reloc will refer to a symbol as usual. It will direct the program linker to add a `PLT` entry for that symbol, if one does not already exist. The computation of the reloc is then a PC-relative reference to the `PLT` entry. (The 32 in the name of the reloc refers to the size of the reference, which is 32 bits). Yesterday I described how on the `i386` every `PLT` entry also has a corresponding `GOT` entry, so the `R_386_PLT32` reloc actually directs the program linker to create both a `PLT` entry and a `GOT` entry.

When the program linker creates an entry in the `PLT` or the `GOT`, it must also generate a dynamic reloc to tell the dynamic linker about the entry. This will typically be a `JMP_SLOT` or `GLOB_DAT` relocation.

This all means that the program linker must keep track of the `PLT` entry and the `GOT` entry for each symbol. Initially, of course,

there will be no such entries. When the linker sees a PLT or GOT reloc, it must check whether the symbol referenced by the reloc already has a PLT or GOT entry, and create one if it does not. Note that it is possible for a single symbol to have both a PLT entry and a GOT entry; this will happen for position independent code which both calls a function and also takes its address.

The dynamic linker's job for the PLT and GOT tables is to simply compute the `JMP_SLOT` and `GLOB_DAT` relocations at runtime. The main complexity here is the lazy evaluation of PLT entries which I described yesterday.

The fact that C permits taking the address of a function introduces an interesting wrinkle. In C you are permitted to take the address of a function, and you are permitted to compare that address to another function address. The problem is that if you take the address of a function in a shared library, the natural result would be to get the address of the PLT entry. After all, that is address to which a call to the function will jump. However, each shared library has its own PLT, and thus the address of a particular function would differ in each shared library. That means that comparisons of function pointers generated in different shared libraries may be different when they should be the same. This is not a purely hypothetical problem; when I did a port which got it wrong, before I fixed the bug I saw failures in the Tcl shared library when it compared function pointers.

The fix for this bug on most processors is a special marking for a symbol which has a PLT entry but is not defined. Typically the symbol will be marked as undefined, but with a non-zero value—the value will be set to the address of the PLT entry. When the dynamic linker is searching for the value of a symbol to use for a reloc other than a `JMP_SLOT` reloc, if it finds such a specially marked symbol, it will use the non-zero value. This will ensure that all references to the symbol which are not function calls will use the same value. To make this work, the compiler and assembler must make sure that any reference to a function which does not involve calling it will not carry a standard PLT reloc. This special handling of function addresses needs to be implemented in both the program linker and the dynamic linker.

ELF Symbols

OK, enough about shared libraries. Let's go over ELF symbols in more detail. I'm not going to lay out the exact data structures—go to the ELF ABI for that. I'm going to take about the different fields and what they mean. Many of the different types of ELF symbols are also used by other object file formats, but I won't cover that.

An entry in an ELF symbol table has eight pieces of information: a name, a value, a size, a section, a binding, a type, a visibility, and undefined additional information (currently there are six undefined bits, though more may be added). An ELF symbol

defined in a shared object may also have an associated version name.

The name is obvious.

For an ordinary defined symbol, the section is some section in the file (specifically, the symbol table entry holds an index into the section table). For an object file the value is relative to the start of the section. For an executable the value is an absolute address. For a shared library the value is relative to the base address.

For an undefined reference symbol, the section index is the special value `SHN_UNDEF` which has the value 0. A section index of `SHN_ABS` (`0xffff1`) indicates that the value of the symbol is an absolute value, not relative to any section.

A section index of `SHN_COMMON` (`0xffff2`) indicates a common symbol. Common symbols were invented to handle Fortran common blocks, and they are also often used for uninitialized global variables in C. A common symbol has unusual semantics. Common symbols have a value of zero, but set the size field to the desired size. If one object file has a common symbol and another has a definition, the common symbol is treated as an undefined reference. If there is no definition for a common symbol, the program linker acts as though it saw a definition initialized to zero of the appropriate size. Two object files may have common symbols of different sizes, in which case the

program linker will use the largest size. Implementing common symbol semantics across shared libraries is a touchy subject, somewhat helped by the recent introduction of a type for common symbols as well as a special section index (see the discussion of symbol types below).

The size of an ELF symbol, other than a common symbol, is the size of the variable or function. This is mainly used for debugging purposes.

The binding of an elf symbol is global, local, or weak. A global symbol is globally visible. A local symbol is only locally visible (e.g., a static function). Weak symbols come in two flavors. A weak undefined reference is like an ordinary undefined reference, except that it is not an error if a relocation refers to a weak undefined reference symbol which has no defining symbol. Instead, the relocation is computed as though the symbol had the value zero.

A weak defined symbol is permitted to be linked with a non-weak defined symbol of the same name without causing a multiple definition error. Historically there are two ways for the program linker to handle a weak defined symbol. On SVR4 if the program linker sees a weak defined symbol followed by a non-weak defined symbol with the same name, it will issue a multiple definition error. However, a non-weak defined symbol followed by a weak defined symbol will not cause an error. On Solaris, a weak defined symbol followed by a non-weak defined symbol is

handled by causing all references to attach to the non-weak defined symbol, with no error. This difference in behaviour is due to an ambiguity in the ELF ABI which was read differently by different people. The GNU linker follows the Solaris behaviour.

The type of an ELF symbol is one of the following:

- `STT_NOTYPE`: no particular type.
- `STT_OBJECT`: a data object, such as a variable.
- `STT_FUNC`: a function
- `STT_SECTION`: a local symbol associated with a section. This type of symbol is used to reduce the number of local symbols required, by changing all relocations against local symbols in a specific section to use the `STT_SECTION` symbol instead.
- `STT_FILE`: a special symbol whose name is the name of the source file which produced the object file.
- `STT_COMMON`: a common symbol. This is the same as setting the section index to `SHN_COMMON`, except in a shared object. The program linker will normally have allocated space for the common symbol in the shared object, so it will have a real section index. The `STT_COMMON` type tells the dynamic linker that although the symbol has a regular definition, it is a common symbol.
- `STT_TLS`: a symbol in the Thread Local Storage area. I will describe this in more detail some other day.

ELF symbol visibility was invented to provide more control over which symbols were accessible outside a shared library. The basic idea is that a symbol may be global within a shared library, but local outside the shared library.

- STV_DEFAULT: the usual visibility rules apply: global symbols are visible everywhere.
- STV_INTERNAL: the symbol is not accessible outside the current executable or shared library.
- STV_HIDDEN: the symbol is not visible outside the current executable or shared library, but it may be accessed indirectly, probably because some code took its address.
- STV_PROTECTED: the symbol is visible outside the current executable or shared object, but it may not be overridden. That is, if a protected symbol in a shared library is referenced by other code in the shared library, that other code will always reference the symbol in the shared library, even if the executable defines a symbol with the same name.

I'll described symbol versions later.

More tomorrow.

Posted August 28, 2007 in [Programming](#)
by Ian Lance Taylor

Tags:

Comments

11 responses to “Linkers part 5”

[christian schorn » Blog Archive » links for 2007-08-30](#)
[August 30, 2007](#)

[...] Airs – Ian Lance Taylor Â» Linkers part 5 (tags: programming basics) [...]



lev

[September 19, 2007](#)

I'm finding this series of posts on linkers very interesting, and mostly very clear. I have a couple of questions regarding this one, though.

1) I understand why it's wrong giving the address of the PLT entry when C code takes the address of a function. But you say that the way around this is to specially mark such uses of a function, with a special symbol that has the value of the address of the PLT entry. Isn't this the same thing? I'm obviously not following that paragraph properly.

2) What possible difference can it make to the linker whether a symbol is marked STV_INTERNAL vs STV_HIDDEN? I can understand that the compiler might be able to do some optimizations if it knows that the function will never be called from outside the executable/shared lib — maybe can avoid

loading the PIC register since you know it's already done by the caller. But that's the compiler: why would the linker need to know the difference between internal and hidden?

Thanks for some interesting articles.



[Ian Lance Taylor](#)

[September 19, 2007](#)

Thanks for the note.

It's OK to make the address of the function be the address of the PLT entry, what matters is that every reference to the function, no matter where it occurs, get that same address. So there is a two-step process. First, the program linker marks the dynamic symbol in a special way by giving it a non-zero value A, and it also uses A for any relocations which reference the function. The dynamic linker then makes sure to use A for any reference to the function other than actually calling it. That is, in the main executable, A is used for any reloc other than a PLT reloc. And likewise in a shared library (typically the only other reloc would be a GOT reloc). This ensures that every reference to the function, other than calling it, gets the value A. Thus comparisons of the function address for equality will work.

Note that this is not an issue for a function defined in the executable. In that case the dynamic linker will always use the address in the executable for all references to the function. This

is also not an issue for a function reference from a shared library. The shared library will naturally have dynamic relocations for the function, and the usual dynamic linker algorithm will ensure that all those relocations refer to the same value.

The problem only arises for a function reference from the main executable. In this case there may not be a dynamic relocation for all references to the function, since the program linker will be able to resolve those relocations to the PLT address. So the special non-zero value in the dynamic symbol table records that there was a reference to the function other than calling it, and it tells the dynamic linker that it must use that address when resolving dynamic relocations in shared libraries other than calling the function.

I hope that makes some sense.

Finally, you're right, both the program linker and the dynamic linker should treat internal and hidden symbols exactly the same. Explicitly recording both types in the ELF symbol visibility field is just for information. Actually I don't know of any systems which actually treat internal symbols differently from hidden symbols in any way, though no doubt there are some.



lev

[September 25, 2007](#)

Thanks for the further explanation. That was the clue I needed. It took me a while looking at the assembler that gets generated, but I figured it out. I had the wrong idea about how the relocations worked for the main executable.

This stuff is certainly confusing, at least if you're not used to the proper way of thinking.

As for the difference between internal and hidden, I found this discussion:

http://groups.google.com/group/generic-abi/browse_thread/thread/1a84adc15666164

where Jim Dehnert, apparently the SGI representative who originally requested the addition of STV_INTERNAL to the gABI, posts here:

http://groups.google.com/group/generic-abi/browse_thread/thread/2c3c04f556d9b84d

He can't remember exactly why they needed it, but thinks it was only relevant to link-time (interprocedural) optimization.

Everyone else in that discussion (8 authors) says that they treat STV_HIDDEN and STV_INTERNAL identically.

Finally, if you're not fed up with answering questions about visibility.... In Ulrich Drepper's DSO how-to:

<http://people.redhat.com/drepper/dsohowto.pdf>

Drepper says that protected visibility sounds nice but is even more expensive than default visibility. I can't see why this would be. I see that it would be very tricky if you were allowed use

protected function addresses in a non-call way in the DSO. But the gnu toolchain specifically forbids this. Eg:

```
cmt:~/dso> cat w.c
```

```
void prot(void) __attribute__(( visibility ("protected" )));
```

```
int f(void (*p)(void) )
```

```
{
```

```
return p==prot;
```

```
}
```

```
void prot(void)
```

```
{
```

```
/*nothing*/
```

```
}
```

```
cmt:~/dso> gcc -fpic -o w.so -shared w.c
```

```
/usr/lib/gcc/i586-suse-linux/4.1.0/../../../../i586-suse-linux/bin/ld:
```

```
/tmp/ccsNpSI0.o: relocation R_386_GOTOFF against protected  
function `prot' can not be used when making a shared object
```

```
/usr/lib/gcc/i586-suse-linux/4.1.0/../../../../i586-suse-linux/bin/ld:
```

```
final link failed: Bad value
```

```
collect2: ld returned 1 exit status
```

As long as one can deal with this restriction, shouldn't protected visibility be an optimal solution for both intra-DSO calls (bypassing the dynamic linker and the PLT) and calls and non-call references from outside the DSO (which just use the same

mechanisms as they would with default visibility)? Seems like it has just the same effect as Drepper's suggested:

```
void __attribute__(( visibility("default") )) prot(void)
{
}
extern __typeof(prot) prot_int __attributes__ (( alias("prot"),
visibility("hidden") ));
```

...where you then have to remember to use `prot_int` when referring to the function from within the DSO. The toolchain does allow to take the address of `prot_int` for this method, but you do have to be careful since it won't be same as the address of `prot`.

On the other hand, I'm reluctant to assume that I know any better than Ulrich Drepper about this stuff — he generally seems to know what he's talking about 😊 so... any thoughts? (I spotted some tricky-looking code in glibc's `elf/dl-lookup.c` maybe relating to this, but I don't really follow it).

I'm done reading up to part 17 of your series, and none of the other sections have puzzled me as much as this whole thing about the meaning of symbol visibilities.



[Ian Lance Taylor](#)

[September 25, 2007](#)

Thanks for the comment.

Ulrich is saying that a protected function symbol is expensive because if a shared library references it without calling it, and if the application also references it without calling it, then both references have to return the same address. I personally don't think this is worth worrying about, as the dynamic linker can tell, based on the relocation, whether the function is being called or referenced. This means that a reference rather than a call in a shared library is not optimally efficient. But I don't immediately see why it has to be any more expensive than an ordinary reference to a function in a shared library. In any case, references to functions are not the normal case.

The GNU linker's restriction on using a GOTOFF reloc for a protected function symbol seems to be an attempt to avoid a bug in getting the address of the function. But it seems to be the wrong approach. It should really be marking the GOT entry with an appropriate reloc so that the dynamic linker can resolve it. I don't see any reason that that can not work.

So, yes, I think protected function symbols should work fine, and I don't see any reason to avoid them (modulo toolchain bugs). But I also don't see them as an optimal solution in general. Making a symbol protected changes the semantics: the symbol can no longer be overridden from outside the shared library. If

that is what you want, then fine. But if you want the default semantics, then protected visibility is not helpful.



lev

[September 26, 2007](#)

Thanks for responding.

I think the GNU linker's restriction on using GOTOFF for a protected function symbol is because it would be impossible for the dynamic linker to get it right in all possible cases. When the executable references a function of the same name as the protected symbol, there are possibilities the dynamic linker has to distinguish between: 1) the executable's reference will resolve to the protected function (in which case the reference in the DSO has to be resolved to the executable's PLT address, just as in the default visibility case); 2) the executable's reference will resolve to a different function (in which case the reference can be resolved, for example, to the protected function's load address). Unfortunately at the time of resolving the GOTOFF reference in the DSO, the dynamic linker has no way of choosing between these two possibilities (in particular, it might change in the future, in the presence of `dlopen(..., RTLD_DEEPBIND)` and so on). So, it seems necessary to disallow references to protected symbols.

As for changing the semantics and preventing the symbol from being overridden, this is desirable in the case that Ulrich describes — he's trying to minimize the number of dynamic relocations needed, in order to speed startup of large applications with many libraries. His suggested solution using an internal hidden version of the symbol has the same semantics and also cannot be overridden. I guess I'll ask Ulrich what his concern was with protected symbols.



quietdragon

[November 23, 2008](#)

I think it is also worthy of reference when distinguishing weak from non-weak symbols that the TIS ELF specification says:

- > When the link editor searches archive libraries, it extracts archive
- > members that contain definitions of undefined global symbols. The
- > member's definition may be either a global or a weak symbol. The
- > link editor does not extract archive members to resolve undefined
- > weak symbols. Unresolved weak symbols have a zero value.

The penultimate sentence is key.

[ELF Special Sections | Ben.ZH](#)

[September 27, 2010](#)

[...] st_other: currently holds 0. GNU use it to mark the visibility of the symbol to other components. Its values are 'DEFAULT' 'HIDDEN' 'INTERNAL' and 'PROTECTED'. 'DEFAULT' means the symbol is visible anywhere. Other three described in "GNU Assembler Directives". One googled blog talk about it too, <http://www.airs.com/blog/archives/42> [...]



Ma.Jiang

[May 19, 2011](#)

Thank you Taylor. This is really a very useful article.

But I have a question: why should the address of a function defined in a shared lib be the address of its PLT entry (not the real virtual address of the function)?

I think using the real address is OK, only if all the places used the same value. And I note, on the x86 architecture, if the executable file was compiled with -fpic, the address of the function in libs were their real address.



[Ian Lance Taylor](#)

[May 19, 2011](#)

You're right: using the real address would be OK if all places used the same value. The problem is the executable. The executable is

usually not compiled with -fPIC. That means that the references to function in the code will be compiled to refer to some absolute value that the linker must fill in. Using a dynamic relocation for the code would not be a good idea, as then the code could not be shared. The linker has to use some address that will work as the address of the function, but since the function is (for this example) defined in a shared library, the real address is not known at link time. So the linker uses the address of the PLT entry in the executable. The dynamic linker then has to do the same thing, so that all references use the same value. Hope that makes sense.



Ma.Jiang

[May 19, 2011](#)

Thank you for the answer.

I think i've got your idea: the key problem is that the executable might be compiled without -fPIC.

Thank you again!

Leave a Reply

You must be [logged in](#) to post a comment.

[Airs – Ian Lance Taylor](#)

Proudly powered by [WordPress](#)

| [Section menu](#) | [Main menu](#) |

Airs – Ian Lance Taylor

Linkers part 6

So many things to talk about. Let's go back and cover relocations in some more detail, with some examples.

Relocations

As I said back in part 2, a relocation is a computation to perform on the contents. And as I said yesterday, a relocation can also direct the linker to take other actions, like creating a PLT or GOT entry. Let's take a closer look at the computation.

In general a relocation has a type, a symbol, an offset into the contents, and an addend.

From the linker's point of view, the contents are simply an uninterpreted series of bytes. A relocation changes those bytes as necessary to produce the correct final executable. For example, consider the C code `g = 0;` where `g` is a global variable. On the i386, the compiler will turn this into an assembly language instruction, which will most likely be `movl $0, g` (for position dependent code—position independent code would be loading the address of `g` from the GOT). Now, the `g` in the C code is a global variable, and we all more or less know what that means. The `g` in the assembly code is not that variable. It is a symbol which holds the address of that variable.

The assembler does not know the address of the global variable `g`, which is another way of saying that the assembler does not know the value of the symbol `g`. It is the linker that is going to pick that address. So the assembler has to tell the linker that it needs to use the address of `g` in this instruction. The way the assembler does this is to create a relocation. We don't use a separate relocation type for each instruction; instead, each processor will have a natural set of relocation types which are appropriate for the machine architecture. Each type of relocation expresses a specific computation.

In the i386 case, the assembler will generate these bytes:

```
c7 05 00 00 00 00 00 00 00 00
```

The `c7 05` are the instruction (`movl constant to address`). The first four `00` bytes are the 32-bit constant 0. The second four `00` bytes are the address. The assembler tells the linker to put the value of the symbol `g` into those four bytes by generating (in this case) a `R_386_32` relocation. For this relocation the symbol will be `g`, the offset will be to the last four bytes of the instruction, the type will be `R_386_32`, and the addend will be 0 (in the case of the i386 the addend is stored in the contents rather than in the relocation itself, but this is a detail). The type `R_386_32` expresses a specific computation, which is: put the 32-bit sum of the value of the symbol and the addend into the offset. Since for the i386 the addend is stored in the contents, this can also be expressed as: add the value of the symbol to the 32-bit field at the offset. When

the linker performs this computation, the address in the instruction will be the address of the global variable `g`. Regardless of the details, the important point to note is that the relocation adjusts the contents by applying a specific computation selected by the type.

An example of a simple case which does use an addend would be

```
char a[10]; // A global array.  
char* p = &a[1]; // In a function.
```

The assignment to `p` will wind up requiring a relocation for the symbol `a`. Here the addend will be 1, so that the resulting instruction references `a + 1` rather than `a + 0`.

To point out how relocations are processor dependent, let's consider `g = 0`; on a RISC processor: the PowerPC (in 32-bit mode). In this case, multiple assembly language instructions are required:

```
li 1,0 // Set register 1 to 0  
lis 9,g@ha // Load high-adjusted part of g into register 9  
stw 1,g@l(9) // Store register 1 to address in register 9  
plus low adjusted part g
```

The `lis` instruction loads a value into the upper 16 bits of register 9, setting the lower 16 bits to zero. The `stw` instruction adds a signed 16 bit value to register 9 to form an address, and then stores the value of register 1 at that address. The `@ha` part of the operand directs the assembler to generate a `R_PPC_ADDR16_HA` reloc. The `@l` produces a `R_PPC_ADDR16_LO` reloc. The goal of these relocations is to compute the value of the symbol `g` and use it as the store address.

That is enough information to determine the computations performed by these relocations. The `R_PPC_ADDR16_HA` reloc computes $(\text{SYMBOL} \gg 16) + ((\text{SYMBOL} \& 0x8000) ? 1 : 0)$. The `R_PPC_ADDR16_LO` computes $\text{SYMBOL} \& 0xffff$. The extra computation for `R_PPC_ADDR16_HA` is because the `stw` instruction adds the signed 16-bit value, which means that if the low 16 bits appears negative we have to adjust the high 16 bits accordingly. The offsets of the relocations are such that the 16-bit resulting values are stored into the appropriate parts of the machine instructions.

The specific examples of relocations I've discussed here are ELF specific, but the same sorts of relocations occur for any object file format.

The examples I've shown are for relocations which appear in an object file. As discussed in part 4, these types of relocations may also appear in a shared library, if they are copied there by the program linker. In ELF, there are also specific relocation types

which never appear in object files but only appear in shared libraries or executables. These are the `JMP_SLOT`, `GLOB_DAT`, and `RELATIVE` relocations discussed earlier. Another type of relocation which only appears in an executable is a `COPY` relocation, which I will discuss later.

Position Dependent Shared Libraries

I realized that in part 4 I forgot to say one of the important reasons that ELF shared libraries use PLT and GOT tables. The idea of a shared library is to permit mapping the same shared library into different processes. This only works at maximum efficiency if the shared library code looks the same in each process. If it does not look the same, then each process will need its own private copy, and the savings in physical memory and sharing will be lost.

As discussed in part 4, when the dynamic linker loads a shared library which contains position dependent code, it must apply a set of dynamic relocations. Those relocations will change the code in the shared library, and it will no longer be sharable.

The advantage of the PLT and GOT is that they move the relocations elsewhere, to the PLT and GOT tables themselves. Those tables can then be put into a read-write part of the shared library. This part of the shared library will be much smaller than the code. The PLT and GOT tables will be different in each process using the shared library, but the code will be the same.

I'll be taking a vacation for the long weekend. My next post will most likely be on Tuesday.

Posted August 29, 2007 in [Programming](#)
by Ian Lance Taylor

Tags:

Comments

10 responses to “Linkers part 6”



[ncm](#)

[August 31, 2007](#)

I'm hoping your linker will implement the omit-uncalled-virtuals optimization, as implemented in Symantec's linker years back. Naive linkers see the reference to a virtual function implementation in a virtual function table, itself referenced in a constructor, and link the function even though that function cannot be called by the program. You can tell because that offset into the vtable is never used. You can be smarter: that offset is never used with a static "this" type at or below it in the derivation hierarchy. You can be smarter yet: if the "this" type is below it, and that type or one on the way there provides its own implementation, that can't call yours.

It's tempting to argue that virtual functions are all in shared libraries, these days, or that program size doesn't matter any

more, or that virtual functions aren't so important any more. However, big programs and embedded programs are often linked statically, and cache/VM footprint still matters, and people still insist on making derivation hierarchies.

[Mark J. Wielaard » Ian Lance Taylor's Linker Notes](#)
[August 31, 2007](#)

[...] Linkers part 6 – Relocations, Position Dependent Shared Libraries. [...]



[Ian Lance Taylor](#)
[September 3, 2007](#)

The current GNU linker implemented that optimization for a while, using special relocation types to indicate virtual function calls and the class hierarchy. This information was fed into the garbage collector. This was implemented for eCos. I don't think anybody really uses it, though, and I don't know whether it still works correctly.

Using relocation types was the wrong approach. It should be done using a separate side table in an unloaded section. In any case, this optimization requires cooperation with the compiler. It's not particularly hard to implement in the linker as part of a garbage collector to discard unreferenced sections.



[ncm](#)

[September 4, 2007](#)

The optimization would have to be the default, because nobody would know about it; or, if they did, they'd need apparatus in configure to tell whether it was there and turn it on.

Is there something unsafe about the optimization? I suppose a program could dlopen a library that doesn't construct a type T, but uses a virtual member of T not referenced in the main program. You'd like to get an unresolved-symbol error at dlopen time, then, just as when the library references a regular symbol that's not present. But, you 'd also like to have a way to tell the program linker to retain unused virtuals meant to be available for use by dlopened libraries.



[jlh](#)

[October 28, 2008](#)

In the last paragraph, you say that Position Independent Code moves the relocation targets elsewhere, in GOT and PLT. To be more precise, I would say that the relocations targets are moved to the GOT only, which lives in the read/write segment of the memory. The PLT lives in the read/only segment and leverages the GOT to store the resolved function address.

At least, that's what I know about PLT and GOT internals on i386, but I think it is the same on other architectures.



[Ian Lance Taylor](#)

[October 28, 2008](#)

jlh: The 64-bit PowerPC, for example, uses a different scheme. The PLT is not initialized by ld, and lives in uninitialized writable memory. The R_PPC64_JMP_SLOT reloc refers to the PLT.



AR

[November 17, 2008](#)

Why ld wouldn't fill-in PLT (or GOT) that would be calculated for a given preferred load address?

In case of preferred address matching actual load address, ldd needs not do anything, all symbols are already resolved.

Dynamic linker would, however, have to check several things: the library must be exactly the same as the one used for program linking, it would also have to check if LD_PRELOAD is specified; maybe a few other checks, but in most cases I would expect a match and thus pre-linking would be beneficial.

For PLT in shared libraries themselves, the same thing could be applied.

To avoid address collisions and thus relocations, something similar to windows 'rebase' could be used to rewrite PLT (or GOT) for the given load address for most common libraries.



[Ian Lance Taylor](#)

[November 18, 2008](#)

Thanks for the comment.

This is often called prelinking. On GNU/Linux the prelink tool will do this. It's also possible to do it in ld itself; in fact, I once implemented it in GNU ld for x86, although the patches were sufficiently complex, and the advantage sufficiently small, that I didn't contribute them back. The main advantage that prelink has over ld is that prelink can look at all the libraries at once, but ld necessarily sees them one at a time. To make it work you need to put the shared libraries at nonoverlapping addresses. There are a number of complexities which arise, such as symbols defined in multiple shared libraries. These complexities mean that the dynamic linker still has to do something in some cases.



Jay

[February 26, 2009](#)

When a loader is performing dynamic relocations, it has to work out a physical address for symbol + addend. I'm assuming it does this by looking up a segment and working out what address that segment is loaded at. So does it use the segment containing the symbol's vaddr, or the segment containing (symbol's vaddr + addend)?



[Ian Lance Taylor](#)

[February 26, 2009](#)

Most of the standard PLT and GOT relocations do not use an addend. That said, for those relocations where an addend is used, the dynamic linker will generally look up the symbol value and then apply the addend to the final value.

Leave a Reply

You must be [logged in](#) to post a comment.

[Airs – Ian Lance Taylor](#)

Proudly powered by [WordPress](#)

This article was downloaded by **calibre** from <http://www.airs.com/blog/archives/43>

| [Section menu](#) | [Main menu](#) |

Airs – Ian Lance Taylor

Linkers part 7

As we've seen, what linkers do is basically quite simple, but the details can get complicated. The complexity is because smart programmers can see small optimizations to speed up their programs a little bit, and sometimes the only place those optimizations can be implemented is the linker. Each such optimization makes the linker a little more complicated. At the same time, of course, the linker has to run as fast as possible, since nobody wants to sit around waiting for it to finish. Today I'll talk about a classic small optimization implemented by the linker.

Thread Local Storage

I'll assume you know what a thread is. It is often useful to have a global variable which can take on a different value in each thread (if you don't see why this is useful, just trust me on this). That is, the variable is global to the program, but the specific value is local to the thread. If thread A sets the thread local variable to 1, and thread B then sets it to 2, then code running in thread A will continue to see the value 1 for the variable while code running in thread B sees the value 2. In Posix threads this type of variable can be created via `pthread_key_create` and accessed via `pthread_getspecific` and `pthread_setspecific`.

Those functions work well enough, but making a function call for each access is awkward and inconvenient. It would be more useful if you could just declare a regular global variable and mark it as thread local. That is the idea of Thread Local Storage (TLS), which I believe was invented at Sun. On a system which supports TLS, any global (or static) variable may be annotated with `__thread`. The variable is then thread local.

Clearly this requires support from the compiler. It also requires support from the program linker and the dynamic linker. For maximum efficiency—and why do this if you aren't going to get maximum efficiency?—some kernel support is also needed. The design of TLS on ELF systems fully supports shared libraries, including having multiple shared libraries, and the executable itself, use the same name to refer to a single TLS variable. TLS variables can be initialized. Programs can take the address of a TLS variable, and pass the pointers between threads, so the address of a TLS variable is a dynamic value and must be globally unique.

How is this all implemented? First step: define different storage models for TLS variables.

- *Global Dynamic*: Fully general access to TLS variables from an executable or a shared object.
- *Local Dynamic*: Permits access to a variable which is bound locally within the executable or shared object from which it is referenced. This is true for all static TLS variables, for

example. It is also true for protected symbols—I described those back in part 5.

- *Initial Executable*: Permits access to a variable which is known to be part of the TLS image of the executable. This is true for all TLS variables defined in the executable itself, and for all TLS variables in shared libraries explicitly linked with the executable. This is not true for accesses from a shared library, nor for accesses to TLS variables defined in shared libraries opened by `dlopen`.
- *Local Executable*: Permits access to TLS variables defined in the executable itself.

These storage models are defined in decreasing order of flexibility. Now, for efficiency and simplicity, a compiler which supports TLS will permit the developer to specify the appropriate TLS model to use (with `gcc`, this is done with the `-ftls-model` option, although the Global Dynamic and Local Dynamic models also require using `-fpic`). So, when compiling code which will be in an executable and never be in a shared library, the developer may choose to set the TLS storage model to Initial Executable.

Of course, in practice, developers often do not know where code will be used. And developers may not be aware of the intricacies of TLS models. The program linker, on the other hand, knows whether it is creating an executable or a shared library, and it knows whether the TLS variable is defined locally. So the program linker gets the job of automatically optimizing

references to TLS variables when possible. These references take the form of relocations, and the linker optimizes the references by changing the code in various ways.

The program linker is also responsible for gathering all TLS variables together into a single TLS segment (I'll talk more about segments later, for now think of them as a section). The dynamic linker has to group together the TLS segments of the executable and all included shared libraries, resolve the dynamic TLS relocations, and has to build TLS segments dynamically when `dlopen` is used. The kernel has to make it possible for access to the TLS segments be efficient.

That was all pretty general. Let's do an example, again for i386 ELF. There are three different implementations of i386 ELF TLS; I'm going to look at the gnu implementation. Consider this trivial code:

```
__thread int i;  
int foo() { return i; }
```

In global dynamic mode, this generates i386 assembler code like this:

```
leal i@TLSGD(,%ebx,1), %eax
```

```
call __tls_get_addr@PLT
movl (%eax), %eax
```

Recall from part 4 that `%ebx` holds the address of the GOT table. The first instruction will have a `R_386_TLS_GD` relocation for the variable `i`; the relocation will apply to the offset of the `leal` instruction. When the program linker sees this relocation, it will create two consecutive entries in the GOT table for the TLS variable `i`. The first one will get a `R_386_TLS_DTPMOD32` dynamic relocation, and the second will get a `R_386_TLS_DTPOFF32` dynamic relocation. The dynamic linker will set the `DTPMOD32` GOT entry to hold the *module ID* of the object which defines the variable. The module ID is an index within the dynamic linker's tables which identifies the executable or a specific shared library. The dynamic linker will set the `DTPOFF32` GOT entry to the offset within the TLS segment for that module. The `__tls_get_addr` function will use those values to compute the address (this function also takes care of lazy allocation of TLS variables, which is a further optimization specific to the dynamic linker). Note that `__tls_get_addr` is actually implemented by the dynamic linker itself; it follows that global dynamic TLS variables are not supported (and not necessary) in statically linked executables.

At this point you are probably wondering what is so inefficient about `pthread_getspecific`. The real advantage of TLS shows when you see what the program linker can do. The `leal; call` sequence shown above is canonical: the compiler will always

generate the same sequence to access a TLS variable in global dynamic mode. The program linker takes advantage of that fact. If the program linker sees that the code shown above is going into an executable, it knows that the access does not have to be treated as global dynamic; it can be treated as initial executable. The program linker will actually rewrite the code to look like this:

```
movl %gs:0, %eax  
subl $i@GOTTPOFF(%ebx), %eax
```

Here we see that the TLS system has coopted the `%gs` segment register, with cooperation from the operating system, to point to the TLS segment of the executable. For each processor which supports TLS, some such efficiency hack is made. Since the program linker is building the executable, it builds the TLS segment, and knows the offset of `i` in the segment. The `GOTTPOFF` is not a real relocation; it is created and then resolved within the program linker. It is, of course, the offset from the GOT table to the address of `i` in the TLS segment. The `movl (%eax), %eax` from the original sequence remains to actually load the value of the variable.

Actually, that is what would happen if `i` were not defined in the executable itself. In the example I showed, `i` is defined in the executable, so the program linker can actually go from a global dynamic access all the way to a local executable access. That looks like this:

```
movl %gs:0,%eax  
subl $i@TP0FF,%eax
```

Here `i@TP0FF` is simply the known offset of `i` within the TLS segment. I'm not going to go into why this uses `subl` rather than `addl`; suffice it to say that this is another efficiency hack in the dynamic linker.

If you followed all that, you'll see that when an executable accesses a TLS variable which is defined in that executable, it requires two instructions to compute the address, typically followed by another one to actually load or store the value. That is significantly more efficient than calling `pthread_getspecific`. Admittedly, when a shared library accesses a TLS variable, the result is not much better than `pthread_getspecific`, but it shouldn't be any worse, either. And the code using `__thread` is much easier to write and to read.

That was a real whirlwind tour. There are three separate but related TLS implementations on i386 (known as `sun`, `gnu`, and `gnu2`), and 23 different relocation types are defined. I'm certainly not going to try to describe all the details; I don't know them all in any case. They all exist in the name of efficient access to the TLS variables for a given storage model.

Is TLS worth the additional complexity in the program linker and the dynamic linker? Since those tools are used for every

program, and since the C standard global variable `errno` in particular can be implemented using TLS, the answer is most likely yes.

Posted September 3, 2007 in [Programming](#)
by Ian Lance Taylor

Tags:

Comments

18 responses to “Linkers part 7”



[fche](#)

[September 4, 2007](#)

> Is TLS worth the additional complexity [...] `errno` [...] yes

Is it your sense that real programs check `errno` frequently enough

for this difference to be measurable? I don't recall coming across numbers.



[Ian Lance Taylor](#)

[September 4, 2007](#)

I was thinking not so much that real programs check `errno` frequently enough, as that real multi-threaded programs

frequently call library functions which are required to set errno.

But I don't have any numbers either, I'm just speculating.



[ncm](#)

[September 4, 2007](#)

That a pointer to one thread's errno has the same numeric value as a pointer to some other thread's errno came as a surprise to me. That seems like something not only a lot of extra work to support, but also likely to be unportable to some environments, and furthermore not necessarily what I would want anyway.



[Ian Lance Taylor](#)

[September 4, 2007](#)

No, the pointer to one thread's errno has a different numeric value than the pointer to another thread's errno. The address of a `__thread` variable is globally unique—each thread gets a different address for a `__thread` variable. When I say you can pass the pointer between threads, I mean that thread A can pass the address of a `__thread` variable to thread B, and if thread B makes an assignment through that pointer, thread A will see the result in the `__thread` variable but thread B will not. Hope that makes sense.



avjo

[November 4, 2007](#)

Hi Ian,

Again please allow me to express my gratitude. This series is amazing.

I've got two questions please.

1. I can't understand those '@'-based keywords. Can you please explain how are these keywords constructed ? What is this '@' and what can I put at its sides (I don't remember it being mentioned in my AT&T assembly book) ?

e.g. \$i@TPOFF, \$i@GOTTPOFF(%ebx), i@TLSGD(,%ebx,1),
__tls_get_addr@PLT

2. Another unfamiliar item: %gs:0. what is it ? I can't remember the x86 has a %gs register.. and why does it end with a :0 ?

Thank you so much,

avjo



[Ian Lance Taylor](#)

[November 5, 2007](#)

Thanks for the note.

The '@' keywords are extensions to the existing assembly language. They don't change the assembly, but they tell the assembler which relocation types to generate for the operand to which they are attached. The supported keywords are: PLTOFF (64-bit only), PLT, GOTPLT (64-bit only), GOTOFF, GOTPCREL (64-bit only), TLSGD, TLSLDM, TLSLD (64-bit only), GOTTPOFF, TPOFF, NTPOFF (32-bit only), DTPOFF, GOTNTPOFF (32-bit only), INDNTPOFF (32-bit only), GOT, TLSDESC, TLSCALL.

The %gs register is a segment register. The x86 supports several segment registers. These days they are generally all set to the same value, but in the 80286 days they were used to select different portions of memory for different parts of the program. %gs:0 means address 0 in the segment addressed by the %gs segment register.



avjo

[November 7, 2007](#)

Hi Ian and thanks for the explanation.

Do you know of any online page I can read more about this list of supported keywords ?

Thanks again

~avjo



[Ian Lance Taylor](#)

[November 7, 2007](#)

They don't seem to be in the assembler documentation. I think your best bet would be look at the i386 ELF ABI supplement and at the TLS documentation. Here are some links. Look for the sample assembler code. In general the keywords correlate to specific relocation types.

<http://sco.com/developers/devspecs/>

<http://docs.sun.com/app/docs/doc/817-1984/6mhm7pl2a>

<http://people.redhat.com/drepper/tls.pdf>

<http://www.lsd.ic.unicamp.br/~oliva/writeups/TLS/RFC-TLSDESC-x86.txt>



avjo

[April 20, 2008](#)

Hi Ian,

Is there any reason at all to prefer the
pthread_getspecific/setspecific
library calls over a __thread variable ?

What about embedded systems with relatively old kernels (2.6.10
the
oldest) ?

Thanks!

~avjo



[Ian Lance Taylor](#)

[April 22, 2008](#)

As long as your kernel is 2.6.x, you should be able to use `__thread` variables. The only reason I know to prefer `pthread_getspecific` is that you can pass a destructor routine to `pthread_key_create`, which will be run when a thread exits. I don't think there is any way to run a destructor for a `__thread` variable. In general `__thread` variables are more efficient and should be preferred.



avjo

[April 22, 2008](#)

Thank you.

(PS – I still hope to pre-order your Linkers book one day 😊)



avjo

[September 15, 2008](#)

Hi Ian,

When I'm trying to use `__thread` in an application, I get the following gcc error:

```
error: function-scope ~i™ implicitly auto and declared  
~__thread™
```

(all I did is trying to compile an empty C main with the line
`'__thread int i;'`)

Any idea what is wrong ? (I'm using gcc 4.2.3 (Ubuntu 4.2.3-2ubuntu7) on 2.6.24-19 (ubuntu generic x86_64 kernel) on x86_64 platform...

The compile line is just `'gcc attempt.c'..`

Thank you!

~avjo



[Ian Lance Taylor](#)

[September 16, 2008](#)

`__thread` only works for global or static variables. It sounds like you wrote

```
int main() { __thread int i; }
```

That makes `i` a local variable in `main`, which in C is known as an "auto" variable (from the very old but still supported syntax `"auto int i;"`). A local variable can not be a TLS variable. Or, to put it

another way, local variables are always TLS variables, in the sense that they can only be accessed by a single thread. TLS only makes sense when speaking about variables which can be accessed by multiple threads, which means a global or static variable.



erichtsai

[March 15, 2010](#)

Great blog!

After went through a couple of TLS related documents, I still have questions. It seems to me that, by default, an executable will use IE model to access external TLS variable. With IE model, an executable can access all TLS variables in shared libraries explicitly linked with that executable. So, I think these shared objects can't support lazy binding for this executable any more. In order to support lazy binding, either GD model or dlsym() has to be used. Am I right?

Thanks!

Eric



[Ian Lance Taylor](#)

[March 15, 2010](#)

Thanks for the comment. I guess I'm not sure just what you are saying. It's true that when an executable uses the default IE model to access a TLS variable defined in a shared library, the dynamic linker has to resolve that access at startup time, rather than lazily. This doesn't really affect how the shared libraries access the TLS variable, though; they will continue to use a function call to resolve the address.

Lazy binding is not really a feature of TLS variables. Lazy binding is used for function calls, not variable references. TLS variables do support lazy allocation, which is not quite the same thing. It's true that if an executable refers to a TLS variable, then that variable can not be allocated lazily. But that doesn't really matter, as the allocation of a TLS variable referenced by an executable is essentially free. It simply becomes part of the executable's TLS segment.



ndatta

[January 29, 2011](#)

Hi Ian,

Your blog post series on linkers is very well written, thanks!

I had a couple of questions:

(i) Who populates the %fs or %gs register to point to the start of

the TLS segment each time a thread switch happens? Is this done in the pthread library? Or by the NPTL in the Linux kernel? Or by some other mechanism? How can I programmatically verify the same, if that is at all possible?

(ii) Is it not possible to see the value of the %gs or %fs register in gdb while debugging a program using a __thread variable? I compiled a simple test program that defined a __thread long l; global variable and printed its value in main(). When I set a breakpoint in gdb at main, and then do an "info registers" at the breakpoint, it shows the segment registers ds, es, fs and gs to be zero. This doesn't make sense?! The disassembled code shows this instruction:

```
mov %fs:0xffffffffffff8,%rdx
```

I'm assuming that the negative offset is due to your note about the linker generating a subl instead of an addl. Is this correct? And how does it work with negative offsets anyhow?

Thanks again.



[Ian Lance Taylor](#)

[January 30, 2011](#)

On GNU/Linux, the %fs and %gs registers are saved and restored by the kernel on each thread switch, just as with any other register. When a new thread is created, the NPTL pthread library

uses CLONE_SETTLS to tell the kernel to point %fs or %gs to the area passed in as a parameter.

I'm not sure what you want to programmatically verify, so I'm not sure how to answer that question.

Current versions of gdb will print __thread variables correctly. The values of %fs and %gs are difficult to interpret as they are 16-bit segment registers, and do not store addresses directly. I don't know how to get gdb to provide the address associated with a segment register, nor do I know how to print something like %fs:0 directly.

The TLS works with negative offsets by simply having the NTPL library and the kernel point %gs to the top of the statically allocated TLS area.



ndatta

[January 31, 2011](#)

Great, that clears things up. The CLONE_SETTLS patch description is here: <http://lwn.net/Articles/7603/>.

Leave a Reply

You must be [logged in](#) to post a comment.

[Airs – Ian Lance Taylor](#)

Proudly powered by [WordPress](#)

This article was downloaded by **calibre** from <http://www.airs.com/blog/archives/44>

| [Section menu](#) | [Main menu](#) |

Airs – Ian Lance Taylor

Linkers part 8

ELF Segments

Earlier I said that executable file formats were normally the same as object file formats. That is true for ELF, but with a twist. In ELF, object files are composed of sections: all the data in the file is accessed via the section table. Executables and shared libraries normally contain a section table, which is used by programs like `nm`. But the operating system and the dynamic linker do not use the section table. Instead, they use the segment table, which provides an alternative view of the file.

All the contents of an ELF executable or shared library which are to be loaded into memory are contained within a segment (an object file does not have segments). A segment has a type, some flags, a file offset, a virtual address, a physical address, a file size, a memory size, and an alignment. The file offset points to a contiguous set of bytes which are the contents of the segment, the bytes to load into memory. When the operating system or the dynamic linker loads a file, it will do so by walking through the segments and loading them into memory (typically by using the `mmap` system call). All the information needed by the dynamic

linker—the dynamic relocations, the dynamic symbol table, etc.—are accessed via information stored in special segments.

Although an ELF executable or shared library does not, strictly speaking, require any sections, they normally do have them. The contents of a loadable section will fall entirely within a single segment.

The program linker reads sections from the input object files. It sorts and concatenates them into sections in the output file. It maps all the loadable sections into segments in the output file. It lays out the section contents in the output file segments respecting alignment and access requirements, so that the segments may be mapped directly into memory. The sections are mapped to segments based on the access requirements: normally all the read-only sections are mapped to one segment and all the writable sections are mapped to another segment. The address of the latter segment will be set so that it starts on a separate page in memory, permitting `mmap` to set different permissions on the mapped pages.

The segment flags are a bitmask which define access requirements. The defined flags are `PF_R`, `PF_W`, and `PF_X`, which mean, respectively, that the contents must be made readable, writable, or executable.

The segment virtual address is the memory address at which the segment contents are loaded at runtime. The physical address is

officially undefined, but is often used as the load address when using a system which does not use virtual memory. The file size is the size of the contents in the file. The memory size may be larger than the file size when the segment contains uninitialized data; the extra bytes will be filled with zeroes. The alignment of the segment is mainly informative, as the address is already specified.

The ELF segment types are as follows:

- `PT_NULL`: A null entry in the segment table, which is ignored.
- `PT_LOAD`: A loadable entry in the segment table. The operating system or dynamic linker load all segments of this type. All other segments with contents will have their contents contained completely within a `PT_LOAD` segment.
- `PT_DYNAMIC`: The dynamic segment. This points to a series of dynamic tags which the dynamic linker uses to find the dynamic symbol table, dynamic relocations, and other information that it needs.
- `PT_INTERP`: The interpreter segment. This appears in an executable. The operating system uses it to find the name of the dynamic linker to run for the executable. Normally all executables will have the same interpreter name, but on some operating systems different interpreters are used in different emulation modes.
- `PT_NOTE`: A note segment. This contains system dependent note information which may be used by the operating

system or the dynamic linker. On GNU/Linux systems shared libraries often have a ABI tag note which may be used to specify the minimum version of the kernel which is required for the shared library. The dynamic linker uses this when selecting among different shared libraries.

- PT_SHLIB: This is not used as far as I know.
- PT_PHDR: This indicates the address and size of the segment table. This is not too useful in practice as you have to have already found the segment table before you can find this segment.
- PT_TLS: The TLS segment. This holds the initial values for TLS variables.
- PT_GNU_EH_FRAME (0x6474e550): A GNU extension used to hold a sorted table of unwind information. This table is built by the GNU program linker. It is used by gcc's support library to quickly find the appropriate handler for an exception, without requiring exception frames to be registered when the program start.
- PT_GNU_STACK (0x6474e551): A GNU extension used to indicate whether the stack should be executable. This segment has no contents. The dynamic linker sets the permission of the stack in memory to the permissions of this segment.
- PT_GNU_RELRO (0x6474e552): A GNU extension which tells the dynamic linker to set the given address and size to be read-only after applying dynamic relocations. This is used for const variables which require dynamic relocations.

ELF Sections

Now that we've done segments, let's take a quick look at the details of ELF sections. ELF sections are more complicated than segments, in that there are more types of sections. Every ELF object file, and most ELF executables and shared libraries, have a table of sections. The first entry in the table, section 0, is always a null section.

ELF sections have several fields.

- Name.
- Type. I discuss section types below.
- Flags. I discuss section flags below.
- Address. This is the address of the section. In an object file this is normally zero. In an executable or shared library it is the virtual address. Since executables are normally accessed via segments, this is essentially documentation.
- File offset. This is the offset of the contents within the file.
- Size. The size of the section.
- Link. Depending on the section type, this may hold the index of another section in the section table.
- Info. The meaning of this field depends on the section type.
- Address alignment. This is the required alignment of the section. The program linker uses this when laying out the section in memory.
- Entry size. For sections which hold an array of data, this is the size of one data element.

These are the types of ELF sections which the program linker may see.

- SHT_NULL: A null section. Sections with this type may be ignored.
- SHT_PROGBITS: A section holding bits of the program. This is an ordinary section with contents.
- SHT_SYMTAB: The symbol table. This section actually holds the symbol table itself. The section contents are an array of ELF symbol structures.
- SHT_STRTAB: A string table. This type of section holds null-terminated strings. Sections of this type are used for the names of the symbols and the names of the sections themselves.
- SHT_RELA: A relocation table. The link field holds the index of the section to which these relocations apply. These relocations include addends.
- SHT_HASH: A hash table used by the dynamic linker to speed symbol lookup.
- SHT_DYNAMIC: The dynamic tags used by the dynamic linker. Normally the PT_DYNAMIC segment and the SHT_DYNAMIC section will point to the same contents.
- SHT_NOTE: A note section. This is used in system dependent ways. A loadable SHT_NOTE section will become a PT_NOTE segment.
- SHT_NOBITS: A section which takes up memory space but has no associated contents. This is used for zero-initialized data.

- SHT_REL: A relocation table, like SHT_RELA but the relocations have no addends.
- SHT_SHLIB: This is not used as far as I know.
- SHT_DYNSYM: The dynamic symbol table. Normally the DT_SYMTAB dynamic tag will point to the same contents as this section (I haven't discussed dynamic tags yet, though).
- SHT_INIT_ARRAY: This section holds a table of function addresses which should each be called at program startup time, or, for a shared library, when the library is opened by dlopen.
- SHT_FINI_ARRAY: Like SHT_INIT_ARRAY, but called at program exit time or dlclose time.
- SHT_PREINIT_ARRAY: Like SHT_INIT_ARRAY, but called before any shared libraries are initialized. Normally shared libraries initializers are run before the executable initializers. This section type may only be linked into an executable, not into a shared library.
- SHT_GROUP: This is used to group related sections together, so that the program linker may discard them as a unit when appropriate. Sections of this type may only appear in object files. The contents of this type of section are a flag word followed by a series of section indices.
- SHT_SYMTAB_SHNDX: ELF symbol table entries only provide a 16-bit field for the section index. For a file with more than 65536 sections, a section of this type is created. It holds one 32-bit word for each symbol. If a symbol's section index is

SHN_XINDEX, the real section index may be found by looking in the SHT_SYMTAB_SHNDX section.

- SHT_GNU_LIBLIST (0x6ffffff7): A GNU extension used by the prelinker to hold a list of libraries found by the prelinker.
- SHT_GNU_verdef (0x6ffffffd): A Sun and GNU extension used to hold version definitions (I'll take about symbol versions at some point).
- SHT_GNU_verneed (0x6ffffffe): A Sun and GNU extension used to hold versions required from other shared libraries.
- SHT_GNU_versym (0x6fffffff): A Sun and GNU extension used to hold the versions for each symbol.

These are the types of section flags.

- SHF_WRITE: Section contains writable data.
- SHF_ALLOC: Section contains data which should be part of the loaded program image. For example, this would normally be set for a SHT_PROGBITS section and not set for a SHT_SYMTAB section.
- SHF_EXECINSTR: Section contains executable instructions.
- SHF_MERGE: Section contains constants which the program linker may merge together to save space. The compiler can use this type of section for read-only data whose address is unimportant.
- SHF_STRINGS: In conjunction with SHF_MERGE, this means that the section holds null terminated string constants which may be merged.

- SHF_INFO_LINK: This flag indicates that the info field in the section holds a section index.
- SHF_LINK_ORDER: This flag tells the program linker that when it combines sections, this section must appear in the same relative order as the section in the link field. This can be used to ensure that address tables are built in the expected order.
- SHF_OS_NONCONFORMING: If the program linker sees a section with this flag, and does not understand the type or all other flags, then it must issue an error.
- SHF_GROUP: This section appears in a group (see SHT_GROUP, above).
- SHF_TLS: This section holds TLS data.

Posted September 4, 2007 in [Programming](#)
by Ian Lance Taylor

Tags:

Comments

One response to “Linkers part 8”



ohad

[February 29, 2012](#)

A great read, as always.

If this series was available as a book, I'd pay for it even though it's freely available here.

just because it's awesome.

Leave a Reply

You must be [logged in](#) to post a comment.

Airs – Ian Lance Taylor

Proudly powered by [WordPress](#)

This article was downloaded by **calibre** from <http://www.airs.com/blog/archives/45>

| [Section menu](#) | [Main menu](#) |

Airs – Ian Lance Taylor

Linkers part 9

Symbol Versions

A shared library provides an API. Since executables are built with a specific set of header files and linked against a specific instance of the shared library, it also provides an ABI. It is desirable to be able to update the shared library independently of the executable. This permits fixing bugs in the shared library, and it also permits the shared library and the executable to be distributed separately. Sometimes an update to the shared library requires changing the API, and sometimes changing the API requires changing the ABI. When the ABI of a shared library changes, it is no longer possible to update the shared library without updating the executable. This is unfortunate.

For example, consider the system C library and the `stat` function. When file systems were upgraded to support 64-bit file offsets, it became necessary to change the type of some of the fields in the `stat` struct. This is a change in the ABI of `stat`. New versions of the system library should provide a `stat` which returns 64-bit values. But old existing executables call `stat` expecting 32-bit values. This could be addressed by using complicated macros in the system header files. But there is a better way.

The better way is symbol *versions*, which were introduced at Sun and extended by the GNU tools. Every shared library may define a set of symbol versions, and assign specific versions to each defined symbol. The versions and symbol assignments are done by a script passed to the program linker when creating the shared library.

When an executable or shared library A is linked against another shared library B, and A refers to a symbol S defined in B with a specific version, the undefined dynamic symbol reference S in A is given the version of the symbol S in B. When the dynamic linker sees that A refers to a specific version of S, it will link it to that specific version in B. If B later introduces a new version of S, this will not affect A, as long as B continues to provide the old version of S.

For example, when `stat` changes, the C library would provide two versions of `stat`, one with the old version (e.g., `LIBC_1.0`), and one with the new version (`LIBC_2.0`). The new version of `stat` would be marked as the default—the program linker would use it to satisfy references to `stat` in object files. Executables linked against the old version would require the `LIBC_1.0` version of `stat`, and would therefore continue to work. Note that it is even possible for both versions of `stat` to be used in a single program, accessed from different shared libraries.

As you can see, the version effectively is part of the name of the symbol. The biggest difference is that a shared library can define

a specific version which is used to satisfy an unversioned reference.

Versions can also be used in an object file (this is a GNU extension to the original Sun implementation). This is useful for specifying versions without requiring a version script. When a symbol name contains the @ character, the string before the @ is the name of the symbol, and the string after the @ is the version. If there are two consecutive @ characters, then this is the default version.

Relaxation

Generally the program linker does not change the contents other than applying relocations. However, there are some optimizations which the program linker can perform at link time. One of them is *relaxation*.

Relaxation is inherently processor specific. It consists of optimizing code sequences which can become smaller or more efficient when final addresses are known. The most common type of relaxation is for call instructions. A processor like the m68k supports different PC relative call instructions: one with a 16-bit offset, and one with a 32-bit offset. When calling a function which is within range of the 16-bit offset, it is more efficient to use the shorter instruction. The optimization of shrinking these instructions at link time is known as relaxation.

Relaxation is applied based on relocation entries. The linker looks for relocations which may be relaxed, and checks whether they are in range. If they are, the linker applies the relaxation, probably shrinking the size of the contents. The relaxation can normally only be done when the linker recognizes the instruction being relocated. Applying a relaxation may in turn bring other relocations within range, so relaxation is typically done in a loop until there are no more opportunities.

When the linker relaxes a relocation in the middle of a contents, it may need to adjust any PC relative references which cross the point of the relaxation. Therefore, the assembler needs to generate relocation entries for all PC relative references. When not relaxing, these relocations may not be required, as a PC relative reference within a single contents will be valid wherever the contents winds up. When relaxing, though, the linker needs to look through all the other relocations that apply to the contents, and adjust PC relatives one where appropriate. This adjustment will simply consist of recomputing the PC relative offset.

Of course it is also possible to apply relaxations which do not change the size of the contents. For example, on the MIPS the position independent calling sequence is normally to load the address of the function into the \$25 register and then to do an indirect call through the register. When the target of the call is within the 18-bit range of the branch-and-call instruction, it is

normally more efficient to use branch-and-call, since then the processor does not have to wait for the load of \$25 to complete before starting the call. This relaxation changes the instruction sequence without changing the size.

More tomorrow. I apologize for the haphazard arrangement of these linker notes. I'm just writing about ideas as I think of them, rather than being organized about that. If I do collect these notes into an essay, I'll try to make them more structured.

Posted September 5, 2007 in [Programming](#)
by Ian Lance Taylor

Tags:

Comments

6 responses to "Linkers part 9"



[tromeey](#)

[September 6, 2007](#)

An essay? You're just some concrete examples and diagrams away from a book.

Or perhaps you should do a Knuth and write gold in the literate style 😊

This series has been excellent.



[Ian Lance Taylor](#)

[September 6, 2007](#)

Thanks, I'm glad you like it.

I tried a bit of Web programming years ago. It's actually really painful since you have to write in three languages simultaneously: Pascal (or C or whatever), Tex, and Web.



d

[September 20, 2007](#)

What are your intentions about the framework for doing relaxation in gold?

On some architectures relaxing function call sequences changes the size, and so does compacting the constant pool (eliminating duplicates) followed by replacing the references to the constant pool.

Being able to do this easily would be great.

Thanks



[Ian Lance Taylor](#)

[September 20, 2007](#)

The truth is that I really haven't thought about it. The framework for linker relaxation in the GNU linker is quite simple, and does support changing the size of the code.

In gold perhaps it would be useful to have some way for the backend to record which relocations it cared about, and have a generic driver to invoke the backend to relax specific relocations. I guess you would need two sets: relocations which might be relaxable, and relocations which might have to change when a relaxation occurs. That's about as much as I've thought about it, though. Clever ideas would certainly be welcome.



avjo

[November 7, 2007](#)

- > The versions and symbol assignments are done by a script
- > passed to the program linker when creating the shared
- > library.

I'm interested to see a living example of this script.

Can you please help me to find it in, say, glibc ?

How is it typically called ? Any noticeable extension ?

Thanks a lot



[Ian Lance Taylor](#)

[November 7, 2007](#)

In the glibc sources, look for files whose names start with “Versions”. Those are all gathered together to form a version script passed to the linker. The exact details of how glibc puts the final version script together look pretty complicated, and involve using sed and the preprocessor, but looking at the files should give you the flavor of what they look like.

Leave a Reply

You must be [logged in](#) to post a comment.

Airs – Ian Lance Taylor

Proudly powered by [WordPress](#)

This article was downloaded by **calibre** from <http://www.airs.com/blog/archives/46>

| [Section menu](#) | [Main menu](#) |

Airs – Ian Lance Taylor

Linkers part 10

Parallel Linking

It is possible to parallelize the linking process somewhat. This can help hide I/O latency and can take better advantage of modern multi-core systems. My intention with gold is to use these ideas to speed up the linking process.

The first area which can be parallelized is reading the symbols and relocation entries of all the input files. The symbols must be processed in order; otherwise, it will be difficult for the linker to resolve multiple definitions correctly. In particular all the symbols which are used before an archive must be fully processed before the archive is processed, or the linker won't know which members of the archive to include in the link (I guess I haven't talked about archives yet). However, despite these ordering requirements, it can be beneficial to do the actual I/O in parallel.

After all the symbols and relocations have been read, the linker must complete the layout of all the input contents. Most of this can not be done in parallel, as setting the location of one type of contents requires knowing the size of all the preceding types of contents. While doing the layout, the linker can determine the

final location in the output file of all the data which needs to be written out.

After layout is complete, the process of reading the contents, applying relocations, and writing the contents to the output file can be fully parallelized. Each input file can be processed separately.

Since the final size of the output file is known after the layout phase, it is possible to use `mmap` for the output file. When not doing relaxation, it is then possible to read the input contents directly into place in the output file, and to relocation them in place. This reduces the number of system calls required, and ideally will permit the operating system to do optimal disk I/O for the output file.

Just a short entry tonight. More tomorrow.

Posted September 6, 2007 in [Programming](#)
by Ian Lance Taylor

Tags:

Comments

Leave a Reply

You must be [logged in](#) to post a comment.

[Airs – Ian Lance Taylor](#)

Proudly powered by [WordPress](#)

| [Section menu](#) | [Main menu](#) |

Airs – Ian Lance Taylor

Linkers part 11

Archives

Archives are a traditional Unix package format. They are created by the `ar` program, and they are normally named with a `.a` extension. Archives are passed to a Unix linker with the `-l` option.

Although the `ar` program is capable of creating an archive from any type of file, it is normally used to put object files into an archive. When it is used in this way, it creates a symbol table for the archive. The symbol table lists all the symbols defined by any object file in the archive, and for each symbol indicates which object file defines it. Originally the symbol table was created by the `ranlib` program, but these days it is always created by `ar` by default (despite this, many Makefiles continue to run `ranlib` unnecessarily).

When the linker sees an archive, it looks at the archive's symbol table. For each symbol the linker checks whether it has seen an undefined reference to that symbol without seeing a definition. If that is the case, it pulls the object file out of the archive and includes it in the link. In other words, the linker pulls in all the

object files which defines symbols which are referenced but not yet defined.

This operation repeats until no more symbols can be defined by the archive. This permits object files in an archive to refer to symbols defined by other object files in the same archive, without worrying about the order in which they appear.

Note that the linker considers an archive in its position on the command line relative to other object files and archives. If an object file appears after an archive on the command line, that archive will not be used to define symbols referenced by the object file.

In general the linker will not include archives if they provide a definition for a common symbol. You will recall that if the linker sees a common symbol followed by a defined symbol with the same name, it will treat the common symbol as an undefined reference. That will only happen if there is some other reason to include the defined symbol in the link; the defined symbol will not be pulled in from the archive.

There was an interesting twist for common symbols in archives on old a.out-based SunOS systems. If the linker saw a common symbol, and then saw a common symbol in an archive, it would not include the object file from the archive, but it would change the size of the common symbol to the size in the archive if that

were larger than the current size. The C library relied on this behaviour when implementing the `stdin` variable.

My next posting should be on Monday.

Posted September 7, 2007 in [Programming](#)
by Ian Lance Taylor

Tags:

Comments

11 responses to "Linkers part 11"



baruch

[October 9, 2007](#)

What is the reason for the order between the archives and the object files? It can make life easier if the order doesn't matter and you can just place all objects and archives on the command line and let the linker sort it all out.

I believe the microsoft linker doesn't care much about order.



[Ian Lance Taylor](#)

[October 9, 2007](#)

Thanks for the note.

I suspect that the original reason for the ordering was just simplicity. In the original Unix linkers, even the archives were searched in order; there was no archive symbol table. The `tsort` program, which can still be found on a Unix system near you, was used to sort the object files so that the ones which satisfied references of objects in the archive were found later in the archive. The `lorder` shell script built a partial order of dependencies, called `tsort` to build the total order, and built the archive in that order.

Now that the ordering has been established, people take advantage of it to interpose libraries, so that you can supply your own definitions of functions overriding the ones in an archive.

Come to think of it, I never got around to discussing interposition of shared libraries. I'll try to remember to do that some day.



baruch

[October 9, 2007](#)

Thanks for the information on how to get the objects and archives automatically sorted. I don't care much about the games that can be played, I just want the simplicity of letting the computer do the work I want it to do with the least amount of work on my part.

FWIW, I'd be happy to beta test your gold linker, the application at my workplace takes several minutes to link, getting it down will be so nice. I'm willing to act as a guinea pig even for an incomplete linker 😊



[Ian Lance Taylor](#)

[October 10, 2007](#)

Keep an eye on the binutils mailing list (see <http://sourceware.org/binutils/>). I'll announce gold there when it is ready to beta test.



avjo

[November 7, 2007](#)

> The C library relied on this behaviour when implementing
> the stdin variable.

Interesting! Can you please elaborate ? Thanks !



[Ian Lance Taylor](#)

[November 7, 2007](#)

Unfortunately, I don't remember the exact details of the SunOS 4 a.out representation of stdin. I remember that until the GNU linker implemented the common symbol handling I described—adjust the size of the common symbol but do not include the

archive member—it did not work correctly. It made sense at the time, but now I would have to look at an old SunOS system to recreate exactly what happens and why.

I remember that it didn't have to work that way. It was just the way that libc.a happened to be implemented.



avjo

[January 11, 2008](#)

Hi Ian,

I have a question about the process of linking archives. Let's say the linker had an undefined symbol A, which it found in an archive, and therefore pulled out the whole object file in which the defined symbol A have resided. So now a whole new object joins the party. A is resolved, which is good.

But what about other symbols that the new object file might have ? E.g. let's say there was another undefined symbol B which was already resolved to a weak symbol, but now, in the new object, there is a strong definition of B. Shouldn't the linker now take the new definition of B instead of the previously resolved now ? I guess it should, but for that, it need to check all symbols of the new object file that was just added. Does it do that ? Or maybe did I miss something here ?

Thanks!

~avjo



[Ian Lance Taylor](#)

[January 11, 2008](#)

Yes, when the new object file is pulled into the link, all its symbols are checked. The new definition of B will take precedence over the previous weak definition of B.

Once the linker decides to pull an object in from an archive, that object is treated as though the user named it on the command line.



avjo

[January 11, 2008](#)

Thanks Ian !



haizaar

[April 30, 2008](#)

“Once the linker decides to pull an object in from an archive, that object is treated as though the user named it on the command line.” Which means that all unresolved symbols from that object (whose probably are even unrelated the my program) will be resolved as well?



[Ian Lance Taylor](#)

[April 30, 2008](#)

Yes: when an object comes in from an archive, any undefined references that it makes must be satisfied. For example, they may be satisfied by pulling in other objects from the same archive.

Leave a Reply

You must be [logged in](#) to post a comment.

[Airs – Ian Lance Taylor](#)

Proudly powered by [WordPress](#)

This article was downloaded by **calibre** from <http://www.airs.com/blog/archives/48>

| [Section menu](#) | [Main menu](#) |

Airs – Ian Lance Taylor

Linkers part 12

I apologize for the pause in posts. We moved over the weekend. Last Friday at&t told me that the new DSL was working at our new house. However, it did not actually start working outside the house until Wednesday. Then a problem with the internal wiring meant that it was not working inside the house until today. I am now finally back online at home.

Symbol Resolution

I find that symbol resolution is one of the trickier aspects of a linker. Symbol resolution is what the linker does the second and subsequent times that it sees a particular symbol. I've already touched on the topic in a few previous entries, but let's look at it in a bit more depth.

Some symbols are local to a specific object files. We can ignore these for the purposes of symbol resolution, as by definition the linker will never see them more than once. In ELF these are the symbols with a binding of `STB_LOCAL`.

In general, symbols are resolved by name: every symbol with the same name is the same entity. We've already seen a few exceptions to that general rule. A symbol can have a version: two

symbols with the same name but different versions are different symbols. A symbol can have non-default visibility: a symbol with hidden visibility in one shared library is not the same as a symbol with the same name in a different shared library.

The characteristics of a symbol which matter for resolution are:

- The symbol name
- The symbol version.
- Whether the symbol is the default version or not.
- Whether the symbol is a definition or a reference or a common symbol.
- The symbol visibility.
- Whether the symbol is weak or strong (i.e., non-weak).
- Whether the symbol is defined in a regular object file being included in the output, or in a shared library.
- Whether the symbol is thread local.
- Whether the symbol refers to a function or a variable.

The goal of symbol resolution is to determine the final value of the symbol. After all symbols are resolved, we should know the specific object file or shared library which defines the symbol, and we should know the symbol's type, size, etc. It is possible that some symbols will remain undefined after all the symbol tables have been read; in general this is only an error if some relocation refers to that symbol.

At this point I'd like to present a simple algorithm for symbol resolution, but I don't think I can. I'll try to hit all the high points, though. Let's assume that we have two symbols with the same name. Let's call the symbol we saw first A and the new symbol B. (I'm going to ignore symbol visibility in the algorithm below; the effects of visibility should be obvious, I hope.)

1. If A has a version:

- If B has a version different from A, they are actually different symbols.
- If B has the same version as A, they are the same symbol; carry on.
- If B does not have a version, and A is the default version of the symbol, they are the same symbol; carry on.
- Otherwise B is probably a different symbol. But note that if A and B are both undefined references, then it is possible that A refers to the default version of the symbol but we don't yet know that. In that case, if B does not have a version, A and B really are the same symbol. We can't tell until we see the actual definition.

2. If A does not have a version:

- If B does not have a version, they are the same symbol; carry on.
- If B has a version, and it is the default version, they are the same symbol; carry on.
- Otherwise, B is probably a different symbol, as above.

3. If A is thread local and B is not, or vice-versa, then we have an error.
4. If A is an undefined reference:
 - If B is an undefined reference, then we can complete the resolution, and more or less ignore B.
 - If B is a definition or a common symbol, then we can resolve A to B.
5. If A is a strong definition in an object file:
 - If B is an undefined reference, then we resolve B to A.
 - If B is a strong definition in an object file, then we have a multiple definition error.
 - If B is a weak definition in an object file, then A overrides B. In effect, B is ignored.
 - If B is a common symbol, then we treat B as an undefined reference.
 - If B is a definition in a shared library, then A overrides B. The dynamic linker will change all references to B in the shared library to refer to A instead.
6. If A is a weak definition in an object file, we act just like the strong definition case, with one exception: if B is a strong definition in an object file. In the original SVR4 linker, this case was treated as a multiple definition error. In the Solaris and GNU linkers, this case is handled by letting B override A.
7. If A is a common symbol in an object file:
 - If B is a common symbol, we set the size of A to be the maximum of the size of A and the size of B, and then treat B as an undefined reference.

- If B is a definition in a shared library with function type, then A overrides B (this oddball case is required to correctly handle some Unix system libraries).
 - Otherwise, we treat A as an undefined reference.
8. If A is a definition in a shared library, then if B is a definition in a regular object (strong or weak), it overrides A. Otherwise we act as though A were defined in an object file.
9. If A is a common symbol in a shared library, we have a funny case. Symbols in shared libraries must have addresses, so they can't be common in the same sense as symbols in an object file. But ELF does permit symbols in a shared library to have the type `STT_COMMON` (this is a relatively recent addition). For purposes of symbol resolution, if A is a common symbol in a shared library, we still treat it as a definition, unless B is also a common symbol. In the latter case, B overrides A, and the size of B is set to the maximum of the size of A and the size of B.

I hope I got all that right.

More tomorrow, assuming the Internet connection holds up.

Posted September 13, 2007 in [Programming](#)
by Ian Lance Taylor

Tags:

Comments

2 responses to “Linkers part 12”



d

[September 14, 2007](#)

Would gold support leaving the debug info in the object files? Sun supported this for stabs (haven't tried it for dwarf, so they might still support it) and it was very useful when you have a log of huge object files, writing the huge debug info sections to the final binary takes time.



[Ian Lance Taylor](#)

[September 14, 2007](#)

Yes, I'm sure gold will support that feature at some point, though it doesn't yet. It doesn't take much to do it in the linker; it's a bit more work in the debugger.

Leave a Reply

You must be [logged in](#) to post a comment.

[Airs – Ian Lance Taylor](#)

Proudly powered by [WordPress](#)

This article was downloaded by **calibre** from <http://www.airs.com/blog/archives/49>

Airs – Ian Lance Taylor

Linkers part 13

Symbol Versions Redux

I've talked about symbol versions from the linker's point of view. I think it's worth discussing them a bit from the user's point of view.

As I've discussed before, symbol versions are an ELF extension designed to solve a specific problem: making it possible to upgrade a shared library without changing existing executables. That is, they provide backward compatibility for shared libraries. There are a number of related problems which symbol versions do not solve. They do not provide forward compatibility for shared libraries: if you upgrade your executable, you may need to upgrade your shared library also (it would be nice to have a feature to build your executable against an older version of the shared library, but that is difficult to implement in practice). They only work at the shared library interface: they do not help with a change to the ABI of a system call, which is at the kernel interface. They do not help with the problem of sharing incompatible versions of a shared library, as may happen when a complex application is built out of several different existing shared libraries which have incompatible dependencies.

Despite these limitations, shared library backward compatibility is an important issue. Using symbol versions to ensure backward compatibility requires a careful and rigorous approach. You must start by applying a version to every symbol. If a symbol in the shared library does not have a version, then it is impossible to change it in a backward compatible fashion. Then you must pay close attention to the ABI of every symbol. If the ABI of a symbol changes for any reason, you must provide a copy which implements the old ABI. That copy should be marked with the original version. The new symbol must be given a new version.

The ABI of a symbol can change in a number of ways. Any change to the parameter types or the return type of a function is an ABI change. Any change in the type of a variable is an ABI change. If a parameter or a return type is a struct or class, then any change in the type of any field is an ABI change—i.e., if a field in a struct points to another struct, and that struct changes, the ABI has changed. If a function is defined to return an instance of an enum, and a new value is added to the enum, that is an ABI change. In other words, even minor changes can be ABI changes. The question you need to ask is: can existing code which has already been compiled continue to use the new symbol with no change? If the answer is no, you have an ABI change, and you must define a new symbol version.

You must be very careful when writing the symbol implementing the old ABI, if you don't just copy the existing code. You must be

certain that it really does implement the old ABI.

There are some special challenges when using C++. Adding a new virtual method to a class can be an ABI change for any function which uses that class. Providing the backward compatible version of the class in such a situation is very awkward—there is no natural way to specify the name and version to use for the virtual table or the RTTI information for the old version.

Naturally, you must never delete any symbols.

Getting all the details correct, and verifying that you got them correct, requires great attention to detail. Unfortunately, I don't know of any tools to help people write correct version scripts, or to verify them. Still, if implemented correctly, the results are good: existing executables will continue to run.

Static Linking vs. Dynamic Linking

There is, of course, another way to ensure that existing executables will continue to run: link them statically, without using any shared libraries. That will limit their ABI issues to the kernel interface, which is normally significantly smaller than the library interface.

There is a performance tradeoff with static linking. A statically linked program does not get the benefit of sharing libraries with other programs executing at the same time. On the other hand,

a statically linked program does not have to pay the performance penalty of position independent code when executing within the library.

Upgrading the shared library is only possible with dynamic linking. Such an upgrade can provide bug fixes and better performance. Also, the dynamic linker can select a version of the shared library appropriate for the specific platform, which can also help performance.

Static linking permits more reliable testing of the program. You only need to worry about kernel changes, not about shared library changes.

Some people argue that dynamic linking is always superior. I think there are benefits on both sides, and which choice is best depends on the specific circumstances.

More on Monday. If you think I should write about any specific linker related topics which have not already been mentioned in the comments, please let me know.

Posted September 14, 2007 in [Programming](#)
by Ian Lance Taylor

Tags:

Comments

2 responses to “Linkers part 13”



[fche](#)

[September 15, 2007](#)

Re symbol versioning, to what extent does it improve on the flexibility provided by simply retaining the old shared library (ABI provider) under its old SONAME, and letting newer libraries use new SONAME's ?



[Ian Lance Taylor](#)

[September 15, 2007](#)

I see two advantages over changing the SONAME.

The first is that changing the SONAME requires providing a complete new copy of the shared library. This takes up more disk space. More importantly, when different executables running at the same time require different versions of the shared library, if you use a different SONAME they will each use a different library, so no sharing will occur. Using symbol versioning, they will each use the same shared library, so it will be shared. I think this is a fairly decisive argument in favor of symbol versions over changing the SONAME.

The second advantage I see, which is less important, is that an executable linked against a new SONAME will require that new

SONAME to be present on the system. Thus there is no forward compatibility at all. Symbol versioning doesn't provide full forward compatibility, but it does provide a limited variant: if your executable happens to not use any symbols with newer versions, it will still run on systems which only have the older version of the shared library.

Leave a Reply

You must be [logged in](#) to post a comment.

Airs – Ian Lance Taylor

Proudly powered by [WordPress](#)

This article was downloaded by **calibre** from <http://www.airs.com/blog/archives/50>

| [Section menu](#) | [Main menu](#) |

Airs – Ian Lance Taylor

Linkers part 14

Link Time Optimization

I've already mentioned some optimizations which are peculiar to the linker: relaxation and garbage collection of unwanted sections. There is another class of optimizations which occur at link time, but are really related to the compiler. The general name for these optimizations is *link time optimization* or *whole program optimization*.

The general idea is that the compiler optimization passes are run at link time. The advantage of running them at link time is that the compiler can then see the entire program. This permits the compiler to perform optimizations which can not be done when sources files are compiled separately. The most obvious such optimization is inlining functions across source files. Another is optimizing the calling sequence for simple functions—e.g., passing more parameters in registers, or knowing that the function will not clobber all registers; this can only be done when the compiler can see all callers of the function. Experience shows that these and other optimizations can bring significant performance benefits.

Generally these optimizations are implemented by having the compiler write a version of its intermediate representation into the object file, or into some parallel file. The intermediate representation will be the parsed version of the source file, and may already have had some local optimizations applied.

Sometimes the object file contains only the compiler intermediate representation, sometimes it also contains the usual object code. In the former case link time optimization is required, in the latter case it is optional.

I know of two typical ways to implement link time optimization. The first approach is for the compiler to provide a pre-linker. The pre-linker examines the object files looking for stored intermediate representation. When it finds some, it runs the link time optimization passes. The second approach is for the linker proper to call back into the compiler when it finds intermediate representation. This is generally done via some sort of plugin API.

Although these optimizations happen at link time, they are not part of the linker proper, at least not as I defined it. When the compiler reads the stored intermediate representation, it will eventually generate an object file, one way or another. The linker proper will then process that object file as usual. These optimizations should be thought of as part of the compiler.

Initialization Code

C++ permits global variables to have constructors and destructors. The global constructors must be run before `main` starts, and the global destructors must be run after `exit` is called. Making this work requires the compiler and the linker to cooperate.

The a.out object file format is rarely used these days, but the GNU a.out linker has an interesting extension. In a.out symbols have a one byte type field. This encodes a bunch of debugging information, and also the section in which the symbol is defined. The a.out object file format only supports three sections—text, data, and bss. Four symbol types are defined as sets: text set, data set, bss set, and absolute set. A symbol with a set type is permitted to be defined multiple times. The GNU linker will not give a multiple definition error, but will instead build a table with all the values of the symbol. The table will start with one word holding the number of entries, and will end with a zero word. In the output file the set symbol will be defined as the address of the start of the table.

For each C++ global constructor, the compiler would generate a symbol named `__CTOR_LIST__` with the text set type. The value of the symbol in the object file would be the global constructor function. The linker would gather together all the `__CTOR_LIST__` functions into a table. The startup code supplied by the compiler would walk down the `__CTOR_LIST__` table and call each function.

Global destructors were handled similarly, with the name `__DTOR_LIST__`.

Anyhow, so much for a.out. In ELF, global constructors are handled in a fairly similar way, but without using magic symbol types. I'll describe what gcc does. An object file which defines a global constructor will include a `.ctors` section. The compiler will arrange to link special object files at the very start and very end of the link. The one at the start of the link will define a symbol for the `.ctors` section; that symbol will wind up at the start of the section. The one at the end of the link will define a symbol for the end of the `.ctors` section. The compiler startup code will walk between the two symbols, calling the constructors. Global destructors work similarly, in a `.dtors` section.

ELF shared libraries work similarly. When the dynamic linker loads a shared library, it will call the function at the `DT_INIT` tag if there is one. By convention the ELF program linker will set this to the function named `_init`, if there is one. Similarly the `DT_FINI` tag is called when a shared library is unloaded, and the program linker will set this to the function named `_fini`.

As I mentioned earlier, there are also `DT_INIT_ARRAY`, `DT_PREINIT_ARRAY`, and `DT_FINI_ARRAY` tags, which are set based on the `SHT_INIT_ARRAY`, `SHT_PREINIT_ARRAY`, and `SHT_FINI_ARRAY` section types. This is a newer approach in ELF, and does not require relying on special symbol names.

More tomorrow.

Posted September 17, 2007 in [Programming](#)
by Ian Lance Taylor

Tags:

Comments

3 responses to “Linkers part 14”



[ncm](#)

[September 18, 2007](#)

It must have been at 1988 or 89 USENIX that John Reiser presented his results speeding up startup of Mentor Graphics's CAE programs, by link editing. Programs were taking ten minutes to start up because, it turned out, most .cc files #included iostream.h. That file defined a static variable, a class object containing a single int and a constructor. The constructor would call an initializer for the iostream library, and then set the value.

John displayed a real-time animation of these assignments, mapping the addresses of the pages touched to a Hilbert curve projected into a window, which showed program start-up touching pages throughout the static data space of the process. His link editor compacted all those ints to one or a few pages, resulting in program-startup time reduced to seconds.

Nowadays, ELF offers better ways to get libraries initialized, but the problem has got more complicated. ISO Standard C++0x seems unlikely to provide any help defining initialization order in the presence of threads. It might not end up defining semantics initializing (and, particularly, destroying) variables in shared libraries loaded after main() starts. It seems certain not to define destruction in cases where libraries are unloaded. There does seem to be support for requiring that a static object, having been destroyed, may be re-constructed in-place for use by other destructors that may depend on it.

(This was the same John Reiser who first ported Unix to the Vax, and who made the C preprocessor indispensable. He now posts at <http://bitwagon.com/>. His program rtdi seems apropos here: it allows a program to link to more than one version of glibc at a time. Other programs found there may be interesting as well.)

[All about ELF format « \\$HOME](#)

[December 30, 2007](#)

[...] Advanced:

<http://www.airs.com/blog/archives/category/programming/page/5/> <http://www.securityfocus.com/infocus/1872>

<http://plan99.net/~mike/blog/2006/08/25/elf-and-program-loading/> <http://plan99.net/~mike/blog/page/2/>

<http://em386.blogspot.com/2006/10/resolving-elf-relocation-name-symbols.html>

http://packetstormsecurity.org/papers/bypass/GOT_Hijack.txt
<http://www.greyhat.ch/lab/downloads/pic.html> <http://www.linux-foundation.org/spec/book/ELF-IA32/ELF-IA32.html#STD.IA32.ABI.4>
http://neuronicimpulses.blogspot.com/2005_12_01_archive.html
[...]

[Linking – WHO'S AWESOME](#)
[December 13, 2023](#)

[...] used for LTO. Typical LTO includes function inlining, dead code elimination, etc. Read more here. We'll instead focus on a concrete and interesting example: trying to inline a comparison
[...]

Leave a Reply

You must be [logged in](#) to post a comment.

[Airs – Ian Lance Taylor](#)

Proudly powered by [WordPress](#)

This article was downloaded by **calibre** from <http://www.airs.com/blog/archives/51>

| [Section menu](#) | [Main menu](#) |

Airs – Ian Lance Taylor

Linkers part 15

COMDAT sections

In C++ there are several constructs which do not clearly live in a single place. Examples are inline functions defined in a header file, virtual tables, and typeinfo objects. There must be only a single instance of each of these constructs in the final linked program (actually we could probably get away with multiple copies of a virtual table, but the others must be unique since it is possible to take their address). Unfortunately, there is not necessarily a single object file in which they should be generated. These types of constructs are sometimes described as having *vague linkage*.

Linkers implement these features by using *COMDAT* sections (there may be other approaches, but this is the only I know of). COMDAT sections are a special type of section. Each COMDAT section has a special string. When the linker sees multiple COMDAT sections with the same special string, it will only keep one of them.

For example, when the C++ compiler sees an inline function `f1` defined in a header file, but the compiler is unable to inline the function in all uses (perhaps because something takes the

address of the function), the compiler will emit `f1` in a COMDAT section associated with the string `f1`. After the linker sees a COMDAT section `f1`, it will discard all subsequent `f1` COMDAT sections.

This obviously raises the possibility that there will be two entirely different inline functions named `f1`, defined in different header files. This would be an invalid C++ program, violating the One Definition Rule (often abbreviated ODR). Unfortunately, if no source file included both header files, the compiler would be unable to diagnose the error. And, unfortunately, the linker would simply discard the duplicate COMDAT sections, and would not notice the error either. This is an area where some improvements are needed (at least in the GNU tools; I don't know whether any other tools diagnose this error correctly).

The Microsoft PE object file format provides COMDAT sections. These sections can be marked so that duplicate COMDAT sections which do not have identical contents cause an error. That is not as helpful as it seems, as different compiler options may cause valid duplicates to have different contents. The string associated with a COMDAT section is stored in the symbol table.

Before I learned about the Microsoft PE format, I introduced a different type of COMDAT sections into the GNU ELF linker, following a suggestion from Jason Merrill. Any section whose name starts with `".gnu.linkonce."` is a COMDAT section. The associated string is simply the section name itself. Thus the

inline function `f1` would be put into the section `".gnu.linkonce.f1"`. This simple implementation works well enough, but it has a flaw in that some functions require data in multiple sections; e.g., the instructions may be in one section and associated static data may be in another section. Since different instances of the inline function may be compiled differently, the linker can not reliably and consistently discard duplicate data (I don't know how the Microsoft linker handles this problem).

Recent versions of ELF introduce section groups. These implement an officially sanctioned version of COMDAT in ELF, and avoid the problem of `".gnu.linkonce"` sections. I described these briefly in an earlier blog entry. A special section of type `SHT_GROUP` contains a list of section indices in the group. The group is retained or discarded as a whole. The string associated with the group is found in the symbol table. Putting the string in the symbol table makes it awkward to retrieve, but since the string is generally the name of a symbol it means that the string only needs to be stored once in the object file; this is a minor optimization for C++ in which symbol names may be very long.

More tomorrow.

Posted September 18, 2007 in [Programming](#)
by Ian Lance Taylor

Tags:

Comments

2 responses to “Linkers part 15”



[tromeey](#)

[September 20, 2007](#)

FWIW, for the compile server I'm looking into a repository-like approach for things that would ordinarily have vague linkage. Or, perhaps I'll generate them once and then link each into the object files requested by the compilation job. The latter approach may be somewhat slower but has the benefit of creating objects with the expected contents.

Luckily all this is a ways off, so I don't have to make any hard decisions soon.



Joe Buck

[October 8, 2007](#)

There's another related feature of most C++ implementations, invented by Stroustrup (or one of his colleagues), used also by g++. Rather than emitting the virtual function table definition in every object file and using COMDAT, it is emitted in the .o file that contains the definition of the first non-inline virtual function. By the one-definition rule there must be only one such file; doing it this way saves considerable space in .o files. COMDAT is used if all of the functions are defined inline or in the class definition.

The typeinfo object for the class is handled in the same way, as are “out-of-line” definitions for virtual functions that are inline.

This optimization sometimes leads to confusing messages from the linker if there is a missing definition for this first virtual function. I recall that Sun’s linker would generate a message saying something like

virtual function table for class Foo is undefined
[hint: see if the first non-inline virtual function of Foo is defined]

while the GNU linker only gave the first message (or would complain about a missing typeinfo object).

Leave a Reply

You must be [logged in](#) to post a comment.

Airs – Ian Lance Taylor

Proudly powered by [WordPress](#)

This article was downloaded by **calibre** from <http://www.airs.com/blog/archives/52>

| [Section menu](#) | [Main menu](#) |

Airs – Ian Lance Taylor

Linkers part 16

C++ Template Instantiation

There is still more C++ fun at link time, though somewhat less related to the linker proper. A C++ program can declare templates, and instantiate them with specific types. Ideally those specific instantiations will only appear once in a program, not once per source file which instantiates the templates. There are a few ways to make this work.

For object file formats which support COMDAT and vague linkage, which I described yesterday, the simplest and most reliable mechanism is for the compiler to generate all the template instantiations required for a source file and put them into the object file. They should be marked as COMDAT, so that the linker discards all but one copy. This ensures that all template instantiations will be available at link time, and that the executable will have only one copy. This is what gcc does by default for systems which support it. The obvious disadvantages are the time required to compile all the duplicate template instantiations and the space they take up in the object files. This is sometimes called the Borland model, as this is what Borland's C++ compiler did.

Another approach is to not generate any of the template instantiations at compile time. Instead, when linking, if we need a template instantiation which is not found, invoke the compiler to build it. This can be done either by running the linker and looking for error messages or by using a linker plugin to handle an undefined symbol error. The difficulties with this approach are to find the source code to compile and to find the right options to pass to the compiler. Typically the source code is placed into a repository file of some sort at compile time, so that it is available at link time. The complexities of getting the compilation steps right are why this approach is not the default. When it works, though, it can be faster than the duplicate instantiation approach. This is sometimes called the Cfront model.

gcc also supports explicit template instantiation, which can be used to control exactly where templates are instantiated. This approach can work if you have complete control over your source code base, and can instantiate all required templates in some central place. This approach is used for gcc's C++ library, `libstdc++`.

C++ defines a keyword `export` which is supposed to permit exporting template definitions in such a way that they can be read back in by the compiler. gcc does not support this keyword. If it worked, it could be a slightly more reliable way of using a repository when using the Cfront model.

Exception Frames

C++ and other languages support exceptions. When an exception is thrown in one function and caught in another, the program needs to reset the stack pointer and registers to the point where the exception is caught. While resetting the stack pointer, the program needs to identify all local variables in the part of the stack being discarded, and run their destructors if any. This process is known as *unwinding* the stack.

The information needed to unwind the stack is normally stored in tables in the program. Supporting library code is used to read the tables and perform the necessary operations. I'm not going to describe the details of those tables here. However, there is a linker optimization which applies to them.

The support libraries need to be able to find the exception tables at runtime when an exception occurs. An exception can be thrown in one shared library and caught in a different shared library, so finding all the required exception tables can be a nontrivial operation. One approach that can be used is to register the exception tables at program startup time or shared library load time. The registration can be done at the right time using the global constructor mechanism.

However, this approach imposes a runtime cost for exceptions, in that it takes longer for the program to start. Therefore, this is not ideal. The linker can optimize this by building tables which can be used to find the exception tables. The tables built by the GNU linker are sorted for fast lookup by the runtime library. The

tables are put into a `PT_GNU_EH_FRAME` segment. The supporting libraries then need a way to look up a segment of this type. This is done via the `dl_iterate_phdr` API provided by the GNU dynamic linker.

Note that if the compiler believes that the linker will generate a `PT_GNU_EH_FRAME` segment, it won't generate the startup code to register the exception tables. Thus the linker must not fail to create this segment.

Since the GNU linker needs to look at the exception tables in order to generate the `PT_GNU_EH_FRAME` segment, it will also optimize by discarding duplicate exception table information.

I know this section is rather short on details. I hope the general idea is clear.

More tomorrow.

Posted September 19, 2007 in [Programming](#)
by Ian Lance Taylor

Tags:

Comments

Leave a Reply

You must be [logged in](#) to post a comment.

[Airs – Ian Lance Taylor](#)

Proudly powered by [WordPress](#)

| [Section menu](#) | [Main menu](#) |

Airs – Ian Lance Taylor

Linkers part 17

Warning Symbols

The GNU linker supports a weird extension to ELF used to issue warnings when symbols are referenced at link time. This was originally implemented for a.out using a special symbol type. For ELF, I implemented it using a special section name.

If you create a section named `.gnu.warning.SYMBOL`, then if and when the linker sees an undefined reference to `SYMBOL`, it will issue a warning. The warning is triggered by seeing an undefined symbol with the right name in an object file. Unlike the warning about an undefined symbol, it is not triggered by seeing a relocation entry. The text of the warning is simply the contents of the `.gnu.warning.SYMBOL` section.

The GNU C library uses this feature to warn about references to symbols like `gets` which are required by standards but are generally considered to be unsafe. This is done by creating a section named `.gnu.warning.gets` in the same object file which defines `gets`.

The GNU linker also supports another type of warning, triggered by sections named `.gnu.warning` (without the symbol name). If an

object file with a section of that name is included in the link, the linker will issue a warning. Again, the text of the warning is simply the contents of the `.gnu.warning` section. I don't know if anybody actually uses this feature.

Short entry today, more tomorrow.

Posted September 20, 2007 in [Programming](#)
by Ian Lance Taylor

Tags:

Comments

One response to "Linkers part 17"



d

[September 23, 2007](#)

Here's some more warnings that could be useful:

- warn if a symbol is only referred to in the object file where it is defined (that way it can be changed into a static)
- warn if the type of a def and of a use are different... not sure how

feasible this is, as the linker normally does not have the information

needed and the type compatibility rules are hairy. Maybe it could use the debug information...

Just my 2 cents...

Leave a Reply

You must be [logged in](#) to post a comment.

Airs – Ian Lance Taylor

Proudly powered by [WordPress](#)

This article was downloaded by **calibre** from <http://www.airs.com/blog/archives/54>

| [Section menu](#) | [Main menu](#) |

Airs – Ian Lance Taylor

Linkers part 18

Incremental Linking

Often a programmer will make change a single source file and recompile and relink the application. A standard linker will need to read all the input objects and libraries in order to regenerate the executable with the change. For a large application, this is a lot of work. If only one input object file changed, it is a lot more work than really needs to be done. One solution is to use an *incremental linker*. An incremental linker makes incremental changes to an existing executable or shared library, rather than rebuilding them from scratch.

I've never actually written or worked on an incremental linker, but the general idea is straightforward enough. When the linker writes the output file, it must attach additional information.

- The linker must create a mapping of object files to areas in the output file, so that an incremental link will know what to remove when replacing an object file.
- The linker must retain all the relocations for each input object which refer to symbols defined in other objects, so that it can reprocess them when symbols change. The linker

should store the relocations mapped by symbol, so that it can quickly find the relevant relocations.

- The linker should leave extra space in the text and data segments, to allow for object files to grow to a limited extent without requiring rewriting the whole executable. It must keep a map of where this extra space is, as it will tend to move over time over the course of incremental links.
- The linker should keep a list of object file timestamps in the output file, so that it can quickly determine which objects have changed.

With this information, the linker can identify which object files have changed since the last time the output file was linked, and replace them in the existing output file. When an object file changes, the linker can identify all the relocations which refer to symbols defined in the object file, and reprocess them.

When an object file gets too large to fit in the available space in a text or data segment, then the linker has the option of creating additional text or data segments at different addresses. This requires some care to ensure that the new code does not collide with the heap, depending upon how the local `malloc` implementation works. Alternatively, the incremental linker could fall back on doing a full link, and allocating more space again.

Incremental linking can greatly speed up the edit/compile/debug cycle. Unfortunately it is not implemented in most common

linkers. Of course an incremental link is not equivalent to a final link, and in particular some linker optimizations are difficult to implement while acting incrementally. An incremental link is really only suitable for use during the development cycle, which is course the time when the speed of the linker is most important.

More on Monday.

Posted September 21, 2007 in [Programming](#)
by Ian Lance Taylor

Tags:

Comments

2 responses to “Linkers part 18”



Joe Buck

[October 8, 2007](#)

If you ever implement this, please use hashes rather than timestamps (at least as an option). Otherwise there are too many ways to break things; if the .o file is older, but different, the incremental linker still needs to run. Luckily I don't have to use ClearCase anymore, where it was easy for time stamps to move backward.



[Ian Lance Taylor](#)

[October 8, 2007](#)

I would be concerned that using hashes would be too slow, since it would force the linker to actually read the input file. Certainly the linker should not only incrementally link newer files; it should incrementally link any file which has changed in any way at all. That is, think of the timestamp as a very high efficiency hash. It would be easy and appropriate to also check that the size hadn't changed, and, on Unix, that the inode was the same.

Leave a Reply

You must be [logged in](#) to post a comment.

[Airs – Ian Lance Taylor](#)

Proudly powered by [WordPress](#)

This article was downloaded by **calibre** from <http://www.airs.com/blog/archives/55>

| [Section menu](#) | [Main menu](#) |

Airs – Ian Lance Taylor

Linkers part 19

I've pretty much run out of linker topics. Unless I think of something new, I'll make tomorrow's post be the last one, for a total of 20.

`__start` and `__stop` Symbols

A quick note about another GNU linker extension. If the linker sees a section in the output file which can be part of a C variable name—the name contains only alphanumeric characters or underscore—the linker will automatically define symbols marking the start and stop of the section. Note that this is not true of most section names, as by convention most section names start with a period. But the name of a section can be any string; it doesn't have to start with a period. And when that happens for section `NAME`, the GNU linker will define the symbols `__start_NAME` and `__stop_NAME` to the address of the beginning and the end of section, respectively.

This is convenient for collecting some information in several different object files, and then referring to it in the code. For example, the GNU C library uses this to keep a list of functions which may be called to free memory. The `__start` and `__stop` symbols are used to walk through the list.

In C code, these symbols should be declared as something like `extern char __start_NAME[]`. For an extern array the value of the symbol and the value of the variable are the same.

Byte Swapping

The new linker I am working on, gold, is written in C++. One of the attractions was to use template specialization to do efficient byte swapping. Any linker which can be used in a cross-compiler needs to be able to swap bytes when writing them out, in order to generate code for a big-endian system while running on a little-endian system, or vice-versa. The GNU linker always stores data into memory a byte at a time, which is unnecessary for a native linker. Measurements from a few years ago showed that this took about 5% of the linker's CPU time. Since the native linker is by far the most common case, it is worth avoiding this penalty.

In C++, this can be done using templates and template specialization. The idea is to write a template for writing out the data. Then provide two specializations of the template, one for a linker of the same endianness and one for a linker of the opposite endianness. Then pick the one to use at compile time. The code looks this; I'm only showing the 16-bit case for simplicity.

```
// Endian simply indicates whether the host is big endian or not.
```

```
struct Endian
{
public:
// Used for template specializations.
static const bool host_big_endian = __BYTE_ORDER ==
__BIG_ENDIAN;
};

// Valtype_base is a template based on size (8, 16, 32, 64)
// which
// defines the type Valtype as the unsigned integer of the
// specified
// size.

template
struct Valtype_base;

template<>
struct Valtype_base<16>
{
typedef uint16_t Valtype;
};

// Convert_endian is a template based on size and on
// whether the host
// and target have the same endianness. It defines the type
// Valtype
// as Valtype_base does, and also defines a function
```

```
convert_host
// which takes an argument of type Valtype and returns the
// same value,
// but swapped if the host and target have different
endianness.
```

```
template
struct Convert_endian;
```

```
template
struct Convert_endian
{
typedef typename Valtype_base::Valtype Valtype;
```

```
static inline Valtype
convert_host(Valtype v)
{ return v; }
};
```

```
template<>
struct Convert_endian<16, false>
{
typedef Valtype_base<16>::Valtype Valtype;
```

```
static inline Valtype
convert_host(Valtype v)
{ return bswap_16(v); }
};
```

```
// Convert is a template based on size and on whether the
target is
// big endian. It defines Valtype and convert_host like
// Convert_endian. That is, it is just like Convert_endian
except in
// the meaning of the second template parameter.
```

```
template
struct Convert
{
typedef typename Valtype_base::Valtype Valtype;

static inline Valtype
convert_host(Valtype v)
{
return Convert_endian
::convert_host(v);
}
};
```

```
// Swap is a template based on size and on whether the
target is big
// endian. It defines the type Valtype and the functions
readval and
// writeval. The functions read and write values of the
appropriate
// size out of buffers, swapping them if necessary.
```

```
template
struct Swap
{
typedef typename Valtype_base::Valtype Valtype;

static inline Valtype
readval(const Valtype* wv)
{ return Convert::convert_host(*wv); }

static inline void
writeval(Valtype* wv, Valtype v)
{ *wv = Convert::convert_host(v); }
};
```

Now, for example, the linker reads a 16-bit big-endian value using `Swap<16,true>::readval`. This works because the linker always knows how much data to swap in, and it always knows whether it is reading big- or little-endian data.

Posted September 24, 2007 in [Programming](#)
by Ian Lance Taylor

Tags:

Comments

3 responses to “Linkers part 19”



alexr

[September 26, 2007](#)

For extra credit, design a specialization that uses the PPC's load/store byte-reversed instructions when accessing a swapped value.



[Ian Lance Taylor](#)

[September 26, 2007](#)

This would be easy enough using a gcc asm statement and extended macro, actually. My code just calls `bswap_16` and friends in the swapped case; on the i386 using glibc that will generate an `rorw` instruction.



pinskia

[March 24, 2008](#)

Alex,

The new builtins that GCC provides will use the PPC's load/store byte-reversed instructions automatically if loading from memory is required. Otherwise it will do the reverse in the register. The new (Cell only) double word byte-reversed instructions support has not been added to GCC yet though. I will get on to adding it when I get some time :).

— Pinski

Leave a Reply

You must be [logged in](#) to post a comment.

Airs – Ian Lance Taylor

Proudly powered by [WordPress](#)

This article was downloaded by **calibre** from <http://www.airs.com/blog/archives/56>

| [Section menu](#) | [Main menu](#) |

Airs – Ian Lance Taylor

Linkers part 20

This will be my last blog posting on linkers for the time being. Tomorrow my blog will return to its usual trivialities. People who are specifically interested in linker information are warned to stop reading with this post.

I'll close the series with a short update on gold, the new linker I've been working on. It currently (September 25, 2007) can create executables. It can not create shared libraries or relocateable objects. It has very limited support for linker scripts—enough to read `/usr/lib/libc.so` on a GNU/Linux system. It doesn't have any interesting new features at this point. It only supports x86. The focus to date has been entirely on speed. It is written to be multi-threaded, but the threading support has not been hooked in yet.

By way of example, when linking a 900M C++ executable, the GNU linker (version 2.16.91 20060118 on an Ubuntu based system) took 700 seconds of user time, 24 seconds of system time, and 16 minutes of wall time. gold took 7 seconds of user time, 3 seconds of system time, and 30 seconds of wall time. So while I can't promise that it will stay as fast as all features are added, it's in a pretty good position at the moment.

I'm the main developer on gold, but I'm not the only person working on it. A few other people are also making improvements.

The goal is to release gold as a free program, ideally as part of the GNU binutils. I want it to be more nearly feature complete before doing this, though. It needs to at least support -shared and -r. I doubt gold will ever support all of the features of the GNU linker. I doubt it will ever support the full GNU linker script language, although I do plan to support enough to link the Linux kernel.

Future plans for gold, once it actually works, include incremental linking and more far-reaching speed improvements.

Posted September 26, 2007 in [Programming](#)
by Ian Lance Taylor

Tags:

Comments

6 responses to "Linkers part 20"



[ncm](#)

[September 27, 2007](#)

I guess now we know the real reason for gold.



avjo

[November 16, 2007](#)

Ian,

Again,

That was a great series of articles. It was a pleasure to read. If it will ever shape into a book, I will no doubt purchase one.

Thanks!

~avgo



[Ian Lance Taylor](#)

[November 16, 2007](#)

Thanks for the note. An actual book is not particularly likely, but, who knows.



[Jeremie LE HEN](#)

[October 28, 2008](#)

I've not yet read everything, but regarding the book I only want to tell my wish.

If you don't want to write a book, I would suggest to contact John Levine to work on a new version on his own book. According to what I've read in your articles, you can bring a couple of

interesting detail that I didn't see in John's book. Moreover, this would relive you from inventing the whole structure of the book.

Undoubtly, John's book is the best documentation one can find on the topic. Your collaboration on a new version on it would certainly bring it to the top 5 computer science books forever ;).



[Ian Lance Taylor](#)

[October 28, 2008](#)

Thanks for the encouragement. Top 5 computer books might be a bit of a stretch, though!



[Sanjoy](#)

[May 25, 2011](#)

Extremely helpful. I'd buy the book too. 😊

Leave a Reply

You must be [logged in](#) to post a comment.

[Airs – Ian Lance Taylor](#)

Proudly powered by [WordPress](#)

This article was downloaded by **calibre** from <http://www.airs.com/blog/archives/57>

| [Section menu](#) | [Main menu](#) |